Procedural automation of general game level generation: the good, the bad and the ugly

Krzysztof Baran*, Caspar Krijgsman†, Steven Lambregts‡, Noah Posner§, Matthijs Wisboom¶
Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology
Van Mourik Broekmanweg 6 2628 XE Delft
Email: *k.p.baran@student.tudelft.nl, †c.j.krijgsman@student.tudelft.nl, ‡s.f.lambregts@student.tudelft.nl, §n.posner@tudelft.nl, ¶m.j.wisboom@student.tudelft.nl
Supervisor: Dr. ir. Rafael Bidarra

***Abstract*—The monumental goal of Artificial Intelligence (AI) is to model general solutions that can be applied to perform a variety of tasks that normally demand human intelligence to solve. Traditionally human game developers painstakingly design and tweak levels until achieving the precise output of their heart's desire. In the gaming industry, AI for level generation can reduce the need for labour-intensive human design. A general game AI for level generation can only be created once we have a method to describe video games. Video Game Description Language (VGDL) is a high-level language for describing 2D arcade games that consists of two parts, a game and level description. Using this language allows us to analyze games at their mechanical level. The problem of General Video Game Level Generation (GVG-LG) can thus be defined as follows: construct a generator that, given a game (e.g. described in VGDL) which can be played by some AI player, builds any required number of different levels for that game which are enjoyable for humans to play [1]. This research investigates the characteristics of what makes automation of general level generation for 2D video games difficult, identifying what exactly makes it so challenging. Solutions such as algorithmic approaches and design patterns shall be presented. By investigating the techniques that have been used so far, empirical evidence will provide key insight into which techniques are most promising to improve level generation quality in the future.**

***Index Terms*—Algorithms, AI, Design Patterns, Game Level Generation, VGDL**

I. Introduction

Pong is thought to be the first video game ever created. It was designed in 1958 [2]. Since then, only more video games have been made, and the levels that were created were specifically designed for individual games. Playing and creating games has become increasingly popular over the last few decades. In a short amount of time, realism and complexity of gameplay have increased drastically. It is because of this complexity that we focus on 2D game levels for this research. 2D game levels are the most interesting for our generation research since these are used in Procedural Content Generation (PCG), which has been very impressive for several years. Although 3D games are also used in PCG and are just as important, we chose to do this research about 2D games, since these are easier to start with.

We conducted this research to have a better insight into the ways of generating content for games in general. Using AI systems to create general content based on an unknown context was something we had not seen before and is still new territory. This research provided us with new insights into 2D video game level generation and taught us about the far-reaching possibilities of AI.
We will focus specifically on current methods for generating games and solving problems using AI. With this focus, we can determine if they can be used in the future for extending the current approach of general video game level generation. Determining what new techniques there are to improve the current methods.

This literature study will answer the following research question: **What makes general level generation of 2D video games so difficult?** We will answer this question by considering the following sub research questions:

- How have these techniques been used in the GVG-LG competition and do they improve the current state of research in general level generation?
- What are the similarities between levels of different 2D video games? Can knowledge of these similarities be used to improve general level generation?
- How can level generation techniques be combined to create better general level generators?

First, we offer a background of the connection between AI, video games, and problem-solving. In section III, we discuss how video games are constructed and what designs overlap in several games. Then in section IV, we describe the current ways of generating, and we talk about the GVG-AI competition. The current situation can be improved, which we will discuss in section V. Finally, we conclude our research and answer our research question in section VI.

II. Background

In order to efficiently solve a problem, it is necessary to understand it. One of the main problems in level generation research is the quantization of the many variables that make level generation hard. There is not a single generally applicable useful heuristic to measure the levels and components. For example, criteria that the level must be enjoyable for the

player who plays the generated level. It is complex to quantify enjoyable levels as this is a non-descriptive and subjective quality of games. On top of that, given the fact that each game has a different set of rules and controls, it is challenging to create levels that are playable, unique, and to some extent, challenging.

We define a playable level to be a level that a player with any experience can win. Researchers have tried to use many methodologies to test the automatically generated levels by creating AI agents first to play and win the level before letting a human play and judge the level. These researchers did not provide definite answers to the level generation problem. However, it helps create a search space of possible solutions and creates a framework for the evaluation of generated levels. The GVG-AI competition is an example of this in practice and is a treasure trove of data. After developing a method to evaluate level quality, it was then possible for research to turn its attention to the level generation problem. Since there is not much support for frameworks to generate levels, researchers at New York University and the University of Essex decided to take it upon themselves to solve the task at hand [3]. Their focus has been directed at level generation, and the main challenge that they found was that designs are made for a specific domain of games. The current human designers have tailored expertise for specific rather than for general 2D games.

A. GVG - AI Competition

The General Video Game AI (GVG-AI) competition is a competition sponsored by DeepMind from Google in which participants try to create AI's that would play and win a wide variety of 2D games [4]. The competition was mostly focused on the General Video Game Playing (GVGP) aspect for multiple years, but lately, there have been two new tracks added: Level and Rule Generation. There have been many different approaches to the competitions every year (GVGP and since recently level generation), and the researchers tried to recreate these results and improve the agents that participated in the competition.

Level generation has been supplemented by adding an interface with features like parameterizing existing games [5]. The automatic level generation has the potential to create many more times the current number of levels than humans can manually produce. Possibly the most challenging form of automatic level generation is when there is a new type of game that has unknown rules before generation (unseen game).

The root problem appears to be describing the game to the agent in a comprehensible manner. One of the ways was to use Design Patterns to help quantify details of the game level classes: solid sprites, collectible sprites, harmful sprites, enemies, and other sprites. Through their modeling, it has been advised to choose design patterns based on their assigned probability value of occurrence in real games. This method increased the variety of design patterns used and the abilities of the 2D game platform, thus improving levels and the games overall [6].

Another method was a Hyper-Heuristic [7] approach where the higher-level agent chooses lower-level agents (existing agents which were the winners of the competitions) which perform well in playing specific games. The hyper-agent learns to make a better selection of these agents. Each of these methods helps us solve the question of how the games should be played and won, which would help us understand how to automate the design of the game levels. They have shown glimpses of promise as they were outperforming the winning agents. There is still a lot to be tested as the GVG-AI competition is new, and solving this playability would implicitly help us with game level generation.

B. GVG - LG Framework

General Video Game Level Generation (GVG-LG) is a Java framework developed by researchers at New York University and the University of Essex. This framework is part of the GVG-AI competition, and studies have been made about the results ever since. The problem of level generation dates back to the problem of the Procedural Content Generation (PCG). It has been a much-researched topic that covered everything from textures generation to game rules creation. The main focus is to remove the human out of the equation and utilize more of the algorithmic approaches like Random, Constructive, and Search-Based Artificial Intelligence. Due to technological developments, that field has been evolving [3]. These technical improvements result in content becoming more sophisticated and computationally expensive and thus create a greater need in video game consumers for new levels, especially in the market of small devices [6].

As mentioned before, the Design Patterns have been one of the most effective tools to help with level generation categorization and design. The problem refers to a standard and recurring design element in object-oriented development, and they give insight to designers about architectural knowledge and provide a template for many situations. The patterns provide a shared vocabulary to name objects and structures that human game designers create and shape, and set rules to express how these building blocks fit together [8]. Unfortunately, a significant challenge of Design Patterns is that there could be many undiscovered patterns as there are countless games, which is significantly more than what the GVG-AI creators provide (a whopping 92 different 2D games) that have to be analyzed [6]. Also, there could be a lot of duplicate Design Patterns, or the patterns might have some attributes that are different, but the game could go under the same name as the sprite design is game dependent. This ambiguity could cause possible conceptual agreement conflicts of the Design Patterns. Design Patterns should be, hence, standardized and agreed upon in order to choose

the right patterns coherently, avoiding double work for the researchers. However, limited research makes it challenging, forming a consensus amongst generating levels. Also, there is minimal documentation for design choices made in games and levels [8].

C. Algorithmic approaches

Since humanity has overcome the technical limitations, researches started writing different algorithms and trying variations of self-defined parameters. Currently, the research in a level generation has limited since level generation is a new field. However, the papers so far tend to use very similar general approaches to tackle the problem, and they have been significant efforts made to make the most optimal generators. Some methods were more promising that other but the reproduced research results show that there is somewhat of an agreement on what is useful and what is not.

1) Constructive: One of the oldest PCG algorithmic approaches is constructive. This method has a structured/iterative approach with specific rules. The algorithm has to abide by these rules. The algorithms utilize a procedural approach that involves adding each element of a level step-by-step [3]. A variation of this is the generate and test technique (also known as search-based). Once the game level is constructed, it is tested whether the level is of a sufficient playability [9]. The difference is that a simple constructive algorithmic method does not perform any improvement iterations and returns an answer without any further consideration. However, it is not very creative and is still semi-random as their requirements are numerical, with few rules on placing components. The study from New York University has shown that players of game levels (from a flight test) cannot spot the difference between Random and Constructive methods, where Random methodology is to place everything in random places until it is a sound level. The constructive algorithmic method has been ranked worse than the random method [10]. Even other studies have shown that this approach does not guarantee achieving the playability of levels every time [10].

2) Search-Based: A Search-Based approach (also known as Generate-and-Test) is a method where each generator uses heuristic functions to check if the current levels generated are improving and pass on some aspects and add mutations to the next generation of levels. This method provides constant improvements in the level based on fitness functions [3]. The parameters that improve continually have to be representative and thus well chosen. One of the approaches is to test the levels' aesthetics (in a mathematical representation) and difficulty. Researchers from Islamabad have shown that it is possible to quantify these metrics and that with their method, levels become more appealing and challenging [9]. The problem is that these tests, while they are promising based on the results, were only executed on a handful of games with very similar game-play.

3) Hyper-Agent: The hyper-Heuristic approach is a top-down technique of choosing the best agent there is based on the context and the "learned" decision tree (learning happens through searching for the features of the game). This area has been researched at New York University and focuses on the playing aspect of GVG-AI, but the technique could have some merit for level generation. The approach is that multiple sub-agents specialize in certain games with specific features. These agents are selected by the hyper-agent that learns about the games through game feature collection later classified by an algorithm. Next, there is the selection process of the sub-agent (low-level agent), which then performs the task at hand (in this case, plays the games). The reason for this approach was that there was no optimal AI agent and play each game well. The lower-level agents are good at specific games but fail in playing general games, while higher-level agents tend to be suitable for general games [7]. Therefore, the researchers opted for a hybrid approach. Their algorithm uses already pre-existing low-level agents while the hyper-agent selects them. The hyper-agents continuously learn by using online playing to help improve the decision-making process about the lower-level agent selection. The authors of this idea claimed that the algorithm outperformed the winners of 2014 and 2015 competitions showing that there is promise in the hyper-heuristic approach, and there are still possibilities for improvement for the future like diversification of 2D games [7]. The reason is that there are very few samples of levels and games the agent can play and learn from the data provided.

III. The problem of level generation for 2D video games

Generating video game levels is one of the oldest ways of algorithmic creation of game content [11]. Each video game has a different type of level to play. This variety makes it hard to create a generator for generating levels that are playable in several games. In the past, developers were limited in the technological possibilities of the hardware devices. These limitations make them use Procedural Content Generation (PCG) techniques to make it possible to create the content for small disk or memory devices. Nowadays, these technological limitations are decreasing, and game level generation is more important than ever for new video games [3].

A. From description to a level

Every Video Game Description Language (VGDL) has several criteria that they have to meet. For example, a VGDL has to be human-readable and extensible [12], which makes sense because, in this way, it should simplify the reading such that it is easily extendable. Once we have a description language, it is possible to start generating game levels that consist of several aspects that are dividable into three components, namely Tile, Graph, and Vector. As the literature shows,

several games were analyzed to create a schema with the division of these three parts [13].

A simple video game is separatable into several components: Map, Objects, Player Definitions, Avatars, Physics, Events, and Rules [12]. These parts are used in most games but applied differently, which makes the levels for these games more unique. Game Definition Language (GDL) is a logic-based language, similar to Prolog [14]. VGDL derives its core from GDL. Although its purpose is to express 2d arcade games [15].

B. Design patterns used for game levels

Design patterns are solutions to recurring design problems [16]. Applying these patterns is particularly popular amongst the Object Orientated Programming Languages community. Localizing instances of these design patterns in existing software produced without explicit use of patterns can improve the maintainability of software. Applying design patterns proved to be helpful. This lead to the development of tools such as the Pat system. These tools help to discover or recover design information. [17]. In the context of 2D video games, design patterns are describable as the in-game tasks that the player needs to solve in order to play successfully. In the gaming community, it is particularly interesting to develop tools that abstract design patterns from VGDL and then generate levels for a game of any general genre. For a level to be enjoyable, it needs to strike an appropriate balance between rewarding and challenging the player [18]. Games often are dissimilar, having very different gameplay objectives and tasks [19]. When generating levels, we should understand that some games share more commonalities among each other than they do with other games. Arcade game genres, such as dungeon-crawlers, horror games, space shooters, first-person shooters, and platformers, have been identified [20]. By using clustering techniques, as shown in figure 1, game genres can be discovered. In the context of general level generation, we can potentially use this idea to take a new unseen game given in VGDL and use a generator that is most suited for that particular genre.

2D video games do share some fundamental characteristics. A game level is traditionally composed of cells, a section of the level where the player can choose a new path. Sharif, Zafar, and Muhammad, describe items placed on cells that fall in one of the following four categories [6].

- Solid Sprites: Blocks the movement of the player.
- Collectible Sprites: Can be destroyed by the player on interaction.
- Harmful Sprites: Are harmful and can kill the player on interaction.
- Enemies: Agents having ammunition and are harmful to the player.

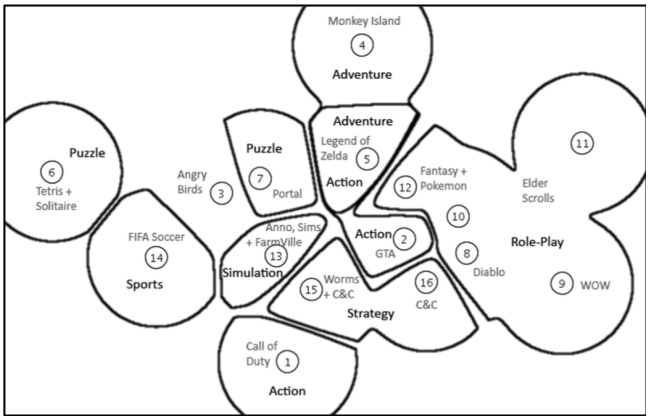

Fig. 1. Illustration of general game genre clustering courtesy of Heintz and Law [21]

Design patterns compartmentalise the features of games into categories and are useful in content generators [22].

C. Procedural Content Generation

Procedural Content Generation (PCG) is the creation of game content with minimal human involvement. Procedural content generators capture game rules as an input and then generate original content for a game [6]. By treating the challenge of level generation as an optimization challenge, fitness functions create the best levels according to some criteria. Algebraic approaches have also created suitable levels by satisfying a set of constraints [23]. One fascinating example of a PCG is that of Rychnovsky's level editor. This editor has the drawback of having no control over the difficulty of the generated levels. This lack of control is because it first generates a level layout and then places game objects in specific locations based on some prior knowledge of the VGDL. After that, Rychnovsky tested level playability by verifying a solution using an algorithm similar to Breadth-First Search (BFS) [10]. In the case of Super Mario Brothers (SMB), it proves that it is possible to notice design patterns by abstracting levels into micro-, meso- and macro-patterns. With some loss of generality, it generates new levels, which replicate the macro-patterns of selected input levels.
Micro-patters occur in video game levels. An example of this is the vertical slices in super Mario in figure 2. Meso-patterns are the gameplay features such as groups of enemies, gaps to jump over, valleys boxing in parts of the level, allowing the player to choose multiple paths and elevating Mario with the aid of stairs. Finally, macro-patterns are sequences of meso-patterns. Usage of micro-patterns is as the building blocks in a search-based PCG approach that searches for macro-patterns. This hierarchy of level abstraction allows for the handling of different aspects of the level generation ranging from low-level detail (micro) to full level overview (macro) [24].

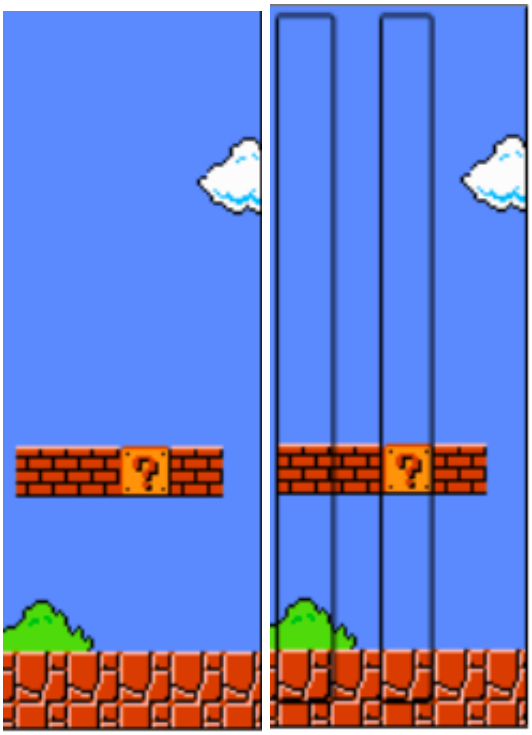

Fig. 2. SMB level 1 and Micro-pattern identification courtesy of Dahlskog & Togelius [24]

D. Graphs

The benefit of using graphs, according to Londoño and Missura, ”is that they enable us to establish semantic relationships between the elements of the levels. Specifically, edges allow us to connect several elements and to give meaning to those connections.” Londoño and Missura have used SMB levels to learn Graph Grammars that can generate new SMB levels. These grammars encode and abstract structural properties of the SMB levels together with their respective probabilities of occurrence. The inferred grammar then creates SMB levels satisfying property constraints [25].

E. Monte Carlo Search Trees

Markov chains train on a corpus of SMB levels. Markov chains proved ”closed” in the sense that they generate levels with similar properties as the input SMB level. Similar to Rychnovsky’s level editor, the main problem with using Markov chains to generate levels is that they offer little to no guarantees about the levels that they generate [26].

F. General Adversarial Network

General Adversarial Networks (GAN) is a specialized Machine Learning class that uses a two-player game where the players are the Generator and Discriminator. The Generator tries to maximize the errors in the data that it is fed and generates new data. The Discriminator, on the other hand, minimizes the errors by trying to identify the fake from the real data. Then through backward propagation, the error weights are increased and decreased respectively [27]. It uses a generative technique which helps to generate new levels based on very few examples given as training data. This technique generates new content using methods such as Latent Variable Evolution (LVE) or Interactive Evolutionary Computation (IEC). A study applying both of these techniques illustrated that a human player (from a flight test) prefers levels that use a mix of these methods, indicating that these techniques generate fun and exciting level designs for two games: Legend of Zelda and Super Mario Brothers [28]. This method seems to be the most promising so far and should be researched further in a wider variety of games. The apparent limitation of this currently is that level generation using this method is only used in the games mentioned above and is not generally applicable. Also, records show that it occasionally generate unwinnable levels due to the game constraints not being followed correctly.

IV. The current state of GVG - LG

The concept of a level generation framework has been around for quite some time. This lead to many people developing a framework for level generators. One of which resulted in the GVG-LG framework. At the moment of writing, the GVG-LG framework seems to be the most reached one. After the development of the GVG-LG framework, the Random, Constructive, and Search-Based Level Generators (SB-LG) made there debut within the framework, of which the SB-LG proved to be the best [6]. As Level Generation is not a fully explored field, the frameworks’ primary use is in the GVG-AI Competition. At this stage, the framework is only able to process single-player, 2D games. The generators are scored based on orientated rankings sorted by scores in the constraints. However, these scores do not always translate to proper levels. Therefore, when it comes to the GVG-AI competition, the level generators are still judged by humans.

A. Methods used

The current explored fields of GVG-LG fall under three categories: **Random** generators, which, as the name suggests, works very simply: randomize a level until it creates a playable one. **Constructive** level generators use a game analyzer to try and place the sprites of the game in strategic locations. Moreover, lastly **Search-Based** (SB) generators use fitness functions based on GVG-AI to determine how challenging a level is.

1) Random Generators: Random generators for game levels rely on a set of probabilities. These probabilities are manually tweaked to provide the best game levels. It starts with an empty map, and for every cell picks a random sprite to place. There are a few conditional rules in the selection: There should always be precisely one avatar, and every sprite has at least one uses.

2) Constructive Generators: The constructive level generator starts by analyzing the *Game Description*. During this analysis, it tries to divide all the sprites into one of five categories, avatar, solid, collectible, harmful, and others. The *Game Analyzer*, used for this analysis, provides additional data on each sprite [3]. Data such as whether the sprite spawns another sprite, and if a sprite is a termination sprite. Data from the analysis is used during the constructive level generators four core states. The core states are *Building a level layout*, *Add an avatar sprite*, *add harmful sprites*, *Add collectibles and other sprites*.

Building a level layout.
If there are solid sprites, the generator takes a type of solid sprite and encloses the level with it. Then based on the calculated solid percentage from the analysis, it fills in the level using solid sprites that are connected. The generator makes sure that it will not block off areas by enclosing it with solid sprites.

Add an avatar sprite.
The generator places a random avatar on a free location.

Add harmful sprites.
Based on the calculated harmful percentage from the analysis, the generator fills in the level using harmful sprites. If a harmful sprite has the label "moving", then the generator will pick a free location away from the avatar. If it is not a moving sprite, it will select any open position.

Add collectibles and other sprites.
Based on the sprites' cover percentage, the generator will place random sprites at free locations.

In its post-processing phase, the generator will make sure that there are enough termination sprites, as stated in the analysis. It will keep adding sprites until this is the case.

Improvements
During the initial study of Khalifa et al., the Random Generator and the Constructive Generator appeared to be indistinguishable [3]. One of the potential problems, as theorized by Dallmeier, is missing collectibles [29]. These missing collectibles could lead to a Constructively Generated level that is unplayable. We encountered two improvements on the Constructive Generator.

Dallmeier himself presented the first improvement. His approach was to add a new step at the end. In this step, the Generator checks if all collectible sprites are present. If not, a missing sprite gets added to the level. In the survey, he conducted it shows that people will now clearly pick the Constructive Generator over the Random Generator [29]. Of the 76 responses, 61 preferred the Constructive Generator over the random generator.

The other improvement came from the 3rd pace contestants of the GVG-LG competition of 2018, Adeel Zafar, who implemented a Pattern-based Generator. This generator upholds specific patterns in the sprites of the level based on the data from the GameAnalyzer. The idea behind this approach is hard to state as we were unable to find a publication about it. We deduced all data on this improvement from the code submitted and the ranking of the GVG-LG competition of 2018. Reaching third place with this approach means that they did improve on the original Constructive Generator. A full analysis of the workings of this algorithm is outside the scope of this paper.

3) Search-Based Generators: For problems where the search space is too large to be explored exhaustively, that means too large to test every possible solution. A search-based algorithm can provide a solution. These algorithms try to find a smart approach to come to a solution. An example of this is Genetic Algorithms. These use a fitness function (or also called cost function) that determines the value of the found solution. For values that are equal or higher than the configured threshold, that solution gets returned.

In the framework provided for the GVG-LG, a Feasible Infeasible 2 Population Genetic Algorithm is presented [30] [3]. This algorithm uses two populations at once, one for feasible chromosomes and one for infeasible chromosomes. The task of the feasible population is to improve the overall fitness for all chromosomes. The infeasible population specializes in decreasing the number of chromosomes that violate the problem constraints. They evolve independently but can exchange children. The Constructive level generator, as described in the previous part, generates the initial populations. The algorithm uses a one-point crossover that swaps two chromosomes around a random tile in the level. The mutation has three operators:

- **Create** a random sprite on a random tile
- **Destroy** all sprites on a random tile
- **Swap** two tiles

For its fitness function, it uses an altered version of the winning agent of the 2014 edition of GVG-AI, *Adrienctx*. It was changed to make it behave a little more like a human player. In addition to *Adrienctx*, the fitness function makes use of *OneStepLookAhead* and *DoNothing*. Both these extra agents will run as many steps as *Adrienctx* did. It also differentiates between two heuristic functions from the feasible population, **Score Difference Fitness**, and **Unique Rule Fitness**. The infeasible population has to adhere to seven different constraints.

- **Avatar Number:** A level can have one avatar
- **Sprite Number:** Must have at least one sprite that is not spawned by other sprites
- **Goal Number:** The goal limit must check out.
- **Cover Percentage:** 5% to 30% of the level must be covered
- **Solution Length:** Solving the level must take at least 200 steps
- **Win:** *Adrienctx* must win

- **Death:** The *DoNothing* must not die at the start and should not succeed in the same amount of steps as *Adrienctx*.

Variations

As an alternative to the Genetic algorithm in the GVG framework, Dallmier did a study on multiple Monte Carlo (MC) methods [29]. The choice to use MC methods is because of their capability to approximate the real value of a state [31]. These values come from pseudo-randomly simulating actions. The results from these actions present the estimated values.

The first method was the **Monte Carlo Tree Search** (MCTS). This method consisted of four phases, *selection*, *expansion*, *rollout*, and *backpropagation* [29], [31]. In the constructed tree, each node contains its value and the number of times it is visited. In the first phase, it selects a node to expand. The selected node gets expanded by an MC rollout until it reaches a terminal state. The terminated state is evaluated and through back-propagation distributed through the tree. Back-propagation will update all the visited nodes from the first phase. For the *selection* phase, the Upper Confidence bound applied to Trees (UCT) is used [32] [29]. The expansion phase uses a method proposed by Coulom. This method adds one node per run [29], [33]. Rollouts are preformed randomly and the results *backpropagated* in the default way [29].

The second method is the **Nested Monte Carlo Search** (NMCS). As the name suggests, it nests levels. It operates on the idea that when an action has a good score on a lower level, it will also score well on a higher level. On its base level, NMCS operates similarly to MCTS. It randomly picks an action until it reaches a final state. From its nested level onward, it recursively searches the lower-level for a legal action to take. From this, it returns the value of the final state. It continues until it can no longer find a better final state. Figure 3 shows an example of the first three iterations. Different from MCTS, NMCS does not rely on building a partial search tree. It memorizes the optimal sequence and result that it has found so far. This sequence gets passed on to the next run. It is passing on the sequence that guarantees that the next run can only be as good or improve, but not degrade. The only parameter for adjusting this algorithm is the number of used nesting levels. Increasing this number increases the runtime, therefore in this project, values from 0 to 3 where considered. Even though NMCS is not tweakable for *exploration* or *exploitation*, it managed to outperform NCTS [29].

The final method is the **Nested Rollout Policy Adaption** (NRPA). Different from the first two methods, this one does not purely use randomness for the rollout phase. This method makes use of policy for determining a series of actions. For any state and any legal action, it computes a code. These codes allow for using domain-specific knowledge. A code will return the same value for a different, yet a sufficiently similar state-action pair. The calculated code provides a policy for the rollout. This policy makes it so that an action is considered

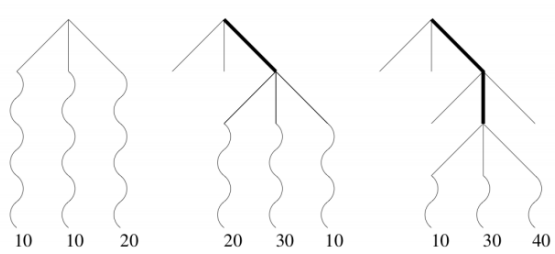

Fig. 3. First three iterations of the NMCS method. The black lines indicate an allowed action. A curly line indicates a lower level search [29]

relatively to the policy value. The remaining structure of this method is similar to that of NMCS, making use of nested levels. The other differences are a specifiable amount of lower-level searches that are initiated for each higher-level search and during the passing on phase. Here the following policy is adopted based on a configured learning rate. As for tweaking for *exploration* or *exploitation*, NRPA does not explicitly contain these parameters. However, picking a high number of lower-level searches results in the same behavior as *exploration*. Picking a low learning rate has the same effect, as it results in a slow convergence [29].

For the evaluation of these methods, an improved version of the *Direct Fitness Function* is used. This version contains more features and does not use *Simulation-based Fitness Functions* to strengthen MC search-based algorithms. The evaluation function employed by Dallmeier uses 11 features consisting of the following: *Accessibility*, *Avatar number*, *Connected walls*, *Cover-percentage*, *Ends initially*, *Goal distance*, *Neutral-harmful ration*, *Simplest avatar*, *Sprite number*, *Space around avatar*, and *Symmetry*. All these features are weighted integer values, resulting in the following formula.

$$f_{directScore} = \frac{\sum_i (w_i f_i)}{\sum_i (w_i)} \tag{1}$$

Where w_i is the weight and f_i the i^{th} feature [29].

In the end, the methods performed well and have proven to be good alternatives to Genetic algorithms. All three methods provided very similar scores, so a definite winner stays undetermined. Dallmeier stated that for future experiments, an improvement upon the evaluation function would be beneficial.

4) Unsupervised Learning: Guzdial and Riedl used an unsupervised process to generate SMB levels from a model trained on SMB gameplay videos. The model could represent probabilistic relationships between shapes and properties. This effect is achievable by learning models of shape-to-shape relationships from gameplay videos. For example generative and probabilistic models As illustrated in the figures 4 and 5 [34].

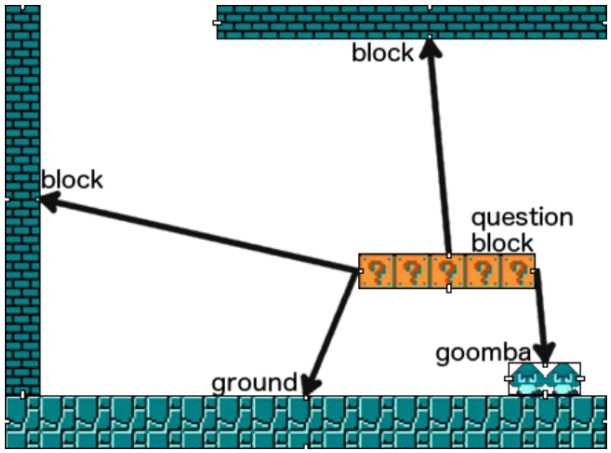

Fig. 4. Example of D node courtesy of Guzdial and Riedl

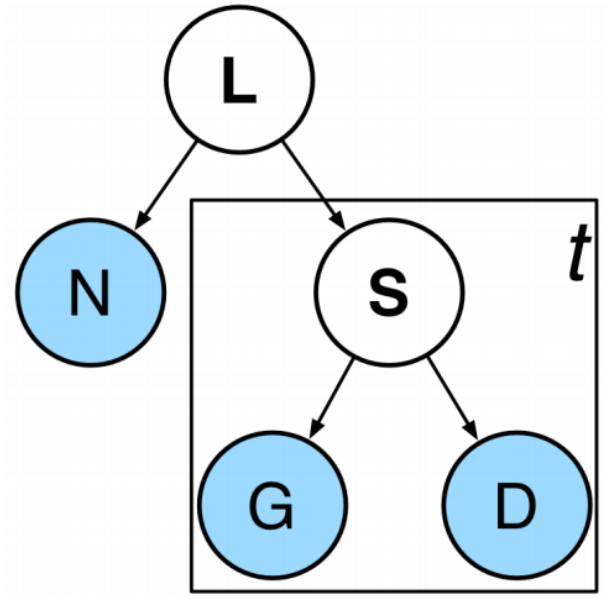

Fig. 5. Visualization of the basic probabilistic model [34]

B. The best out there

At the date of publication, 2018 is the latest version of the GVG-AI competition www.gvgai.net where judges gave a score on the various level generators. As mentioned beforehand, the framework itself will firstly score the generators, after which judges will give a final verdict of the scores. This first score given by the framework can however be taken with a grain of salt, as it does not translate in any way to what the judges will score. A team that had been at number one on the leader-board scored by the framework came in 6^{th} place after being judged by the jury. Again it is difficult for a framework to know what 'Fun' is.

The top-ranking Level Generator of 2018 was of a German student team called 'Architect.' Their structure for a level generator built on the default genetic system provided by the framework. With the difference instead of using a one-point crossover, is uses a two-point crossover and its constructive initialization with a more simplified version of the sample constructive method, almost clinging to a random method [35]. Their methods were far from perfect, however, as they received a jury score of almost 65%. So does this mean their submission was the best generator in general? It all depends on the games. The GVG-AI LG competition only tests with about four different games. Generators that create fairly dense levels might not fare well against games that require the player to move often to complete the level. In the competition for 2016, 2 out of 4 games used to rate the generators required open, sparsely filled playing fields for the game to be even playable. With the next competition in 2018, 2 out of 3 games preformed better by having smaller, more dense levels. Therefore, in constructive generators, to classify the game genre is vital to define the right constraints. For these algorithms, its ability to create a level that is also solvable is still one of the biggest challenges. This challenge might be why Architect's submission won the competition, as, in its first step, the algorithm only fills 10% of the level with sprites apart from the already created layout and position of the player. As it starts relatively small, the search based algorithm can slowly add new sprites if the level is not challenging enough.

The generators in the competition only have a maximum of 5 hours to generate the levels and should, therefore, be as optimized as possible. Here is where the introduction of a tournament based system shows potential. A system making only small changes on a level will take a significant amount of time, especially if the level turns out to be faulty in the end. With a tournament system, the initial level a search-based algorithm can start with will likely show fewer flaws. As it seems at the moment, Constructive and Search-Based algorithms seem to be the best out there. With more experimental approaches like Pattern-based or tournament selections, however, have yet to show their potential.

V. Discussion

As discussed, the main challenges of level generation are accurate benchmarking methods and finding the best construction method for the creation of levels. However, there are many unexplored little details about video games like genres, story-lines, and general design perspectives that could help with better game categorization. Optimal game qualities are hard to quantify as they are subjective. Having these categorizations could potentially help improve design patterns and avoid the problematic duplication of design patterns levels.

The field of Level Generation is a new field with limited research conducted so far. Generative Adversarial Networks (with evolution and exploration methodology) and Hyper-Agent methods have shown the most promise in terms of results. However, there is a tiny variation done in metrics and

tests. We think that more games have to go through GAN level-generation and test on the quality of the levels created. On top of that, we think that also utilization of Hyper-Heuristics could help us develop a more accurate system that would produce quality results every time due to the ability to pick from many specialized AIs and knowing which is the best choice. Most research focuses on game-playing AIs, and it could be used in a level generation as experts suggest there is no ”one-size-fits-all” solution.

The final approach of how we think the system should function is best represented through the work of Kees Fani at Delft University of Technology, as displayed in Figure 6. Even though it is simple, it provides a rough idea of what the combination of GAN and Hyper-Heuristic agents would provide.

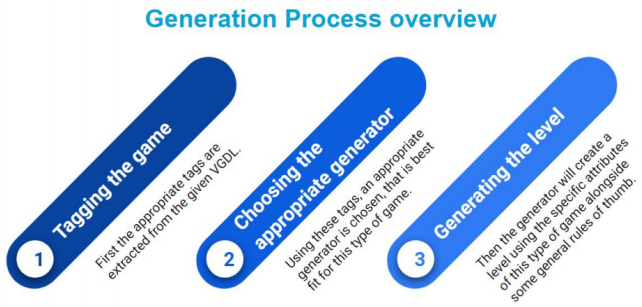

Fig. 6. Generation Process overview courtesy of Kees Fani

In order for the literature study to definitively answer the question **What makes general level generation for 2D video games so difficult?** we answered the following sub research questions:

- **How have techniques been used in the GVG-LG competition and do they improve the current state of research in general level generation?**
  As indicated by the top-ranking Level Generator of 2018, empirical evidence suggests that Genetic Evolutionary algorithms are highly promising. These algorithms commonly employ a fitness function to generate levels from an initial feasible population. The state of the art technique used a constructive initialization with some randomization. The competition winner received a jury score of almost 65%. Thus there is still room to improve upon the best general level generators.

- **What are the similarities between levels of different 2D video games? Can knowledge of these similarities be used to improve general level generation?** In the context of 2D video games, design patterns are the in-game tasks that the player needs to solve in order to play successfully. In the gaming community, it is of particular interest to develop tools that abstract design patterns from VGDL and then generate levels for a game of any general genre. Games often differ from one another greatly [19]. When generating levels, we should understand that some games share more commonalities among each other than they do with other games. Arcade game genres, such as dungeon-crawlers, horror games, space shooters, first-person shooters, and platformers, are identified.

- **How can level generation techniques be combined to create better general level generators?**
  As described by Kees Fani, The best solution on offer may be to combine multiple generators. Each generator is specialized for a specific game genre. The following steps can provide this specialization.

1) Tagging the game with the appropriate tags that are extracted from the arbitrary game given in VGDL.
2) Choosing the most appropriate generator from a set of generators that has been trained previously on a specific type/genre of game.
3) Generate the level by using the most appropriate generator with the specific attributes of the game given in VGDL.

VI. Conclusion

The motivation for this literature study was to learn from the good, the bad, and the ugly of procedural automation. Determining which level generation techniques have been successful from the GVG-AI competition and are promising for future general level generation research. **Constructive and Search-Based algorithms** have shown to perform well when generating levels. Having gained vital insight from answering the sub research questions identified it is now possible to provide an answer to the primary research question posed **What makes general level generation for 2D video games so difficult?** The most challenging test of automatic level generation is when a generator receives a game of an unseen genre in VGDL. The critical challenge is that a game of a new genre is challenging to create levels for since it is not possible to match it to a similar generator that has been pre-trained on a corpus of training data. Similar to specialized human game designers, the literature indicates specialization of level generation research within a particular game/genre such as SMB/platformer. It is harder to create a general generator for a game level that achieves better performance than one that has specialized in that particular one already.

Acknowledgment

We want to express our appreciation to Dr. ir. Rafael Bidarra for his willingness to give his time so generously. Dr. Bidarra’s enthusiastic encouragement, suggestions, and constructive guidance during the planning and development of this research have been immensely valuable.

References

[1] GAIGResearch, “Gaigresearch gvgai,” mar 2015. [Online]. Available: https://github.com/GAIGResearch/GVGAI

[2] I. Flatow, *They All Laughed...: From Light Bulbs to Lasers: The Fascinating Stories Behind the Great Inventions*. HarperCollins, 1993. [Online]. Available: https://books.google.nl/books?id=LSx9lby2bFwC

[3] A. Khalifa, D. Perez-Liebana, S. M. Lucas, and J. Togelius, "General video game level generation," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, ser. GECCO '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 253–259. [Online]. Available: https://doi.org/10.1145/2908812.2908920

[4] D. e. a. Perez-Liebana, mar 2014.

[5] R. D. Gaina, "Gvg framework," feb 2018.

[6] M. Sharif, A. Zafar, and U. Muhammad, "Design patterns and general video game level generation," *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 9, pp. 393–398, 2017.

[7] A. Mendes, J. Togelius, and A. Nealen, "Hyper-heuristic general video game playing," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, Sep. 2016, pp. 1–8.

[8] J. Holopainen and S. Björk, "Game design patterns," *Lecture Notes for GDC*, 2003.

[9] A. Zafar, H. Mujtaba, and M. Beg, "Search-based procedural content generation for gvg-lg," *Applied Soft Computing*, vol. 86, p. 105909, 11 2019.

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

[11] N. Shaker, J. Togelius, and M. J. Nelson, *: Procedural content generation in games: A textbook and an overview of current research*. Procedural Content Generation in Games: A Textbook and an Overview of Current Research, 2015.

[12] M. Ebner, J. Levine, S. M. Lucas, T. Schaul, T. Thompson, and J. Togelius, "Towards a Video Game Description Language," in *Artificial and Computational Intelligence in Games*, ser. Dagstuhl Follow-Ups, S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius, Eds. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, vol. 6, pp. 85–100. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2013/4338

[13] A. J. Summerville, S. Snodgrass, M. Mateas, and S. Ontanón, "The vglc: The video game level corpus," *arXiv preprint arXiv:1606.07487*, 2016.

[14] Stanford, "Game definition language," May 2013.

[15] A. Mora, G. Squillero, A. Agapitos, P. Burelli, W. Bush, S. Cagnoni, C. Cotta, I. Falco, A. Della Cioppa, F. Divina, A. Eiben, A. Esparcia-Alcázar, F. Vega, K. Glette, E. Haasdijk, I. Hidalgo, M. Kampouridis, P. Kaufmann, M. Mavrovouniotis, and M. Zhang, *Applications of Evolutionary Computation - 18th European Conference, EvoApplications*. Springer International Publishing, 01 2015.

[16] L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, and M. Di Penta, "An empirical study on the evolution of design patterns," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 385–394. [Online]. Available: https://doi.org/10.1145/1287624.1287680

[17] C. Kramer and L. Prechelt, "Design recovery by automated search for structural design patterns in object-oriented software," in *Proceedings of WCRE '96: 4rd Working Conference on Reverse Engineering*, 1996, pp. 208–215.

[18] S. Dahlskog and J. Togelius, "Patterns and procedural content generation: revisiting mario in world 1 level 1," in *Proceedings of the First Workshop on Design Patterns in Games*, 2012, pp. 1–8.

[19] S. Björk and J. Holopainen, "Games and design patterns," *The game design reader*, pp. 410–437, 2006.

[20] J. Togelius and G. N. Yannakakis, "General general game ai," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2016, pp. 1–8.

[21] S. Heintz and E. L.-C. Law, "The game genre map: A revised game classification," in *Proceedings of the 2015 Annual Symposium on Computer-Human Interaction in Play*, 2015, pp. 175–184.

[22] S. Dahlskog, S. Björk, and J. Togelius, "Patterns, dungeons and generators," 2015.

[23] G. Smith, "Understanding procedural content generation: a design-centric analysis of the role of pcg in games," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2014, pp. 917–926.

[24] S. Dahlskog and J. Togelius, "A multi-level level generator," in *2014 IEEE Conference on Computational Intelligence and Games*, 2014, pp. 1–8.

[25] S. Londoño and O. Missura, "Graph grammars for super mario bros levels." in *FDG*, 2015.

[26] A. J. Summerville, S. Philip, and M. Mateas, "Mcmcts pcg 4 smb: Monte carlo tree search to guide platformer level generation," in *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[27] J. Rocca. (2019, jan) Understanding generative adversarial networks (gans). [Online]. Available: https://towardsdatascience.com/understanding-generative-adversarial-networks-gans-cd6e4651a29

[28] J. Schrum, J. Gutierrez, V. Volz, J. Liu, S. Lucas, and S. Risi, "Interactive evolution and exploration within latent level-design space of generative adversarial networks," 2020.

[29] E. C. Dallmeier, "Automated level generation for general video games," Aug 2018.

[30] S. O. Kimbrough, G. J. Koehler, M. H. Lu, and D. u. Wood, "On a feasible–infeasible two-population (fi-2pop) genetic algorithm for constrained optimization: Distance tracing and no free lunch," *European Journal of Operational Research*, vol. 190, no. 2, p. 310–327, 2008.

[31] G. Chaslot, M. Winands, J. Uiterwijk, H. V. D. Herik, and B. Bouzy, "Progressive strategies for monte-carlo tree search," *Information Sciences 2007*, p. 655–661, 2007.

[32] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," *Lecture Notes in Computer Science Machine Learning: ECML 2006*, p. 282–293, 2006.

[33] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," *Computers and Games Lecture Notes in Computer Science*, p. 72–83, 2007.

[34] M. Guzdial and M. Riedl, "Game level generation from gameplay videos," in *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2016.

[35] D. Perez-Liebana, S. M. Lucas, R. D. Gaina, J. Togelius, A. Khalifa, and J. Liu, *General Video Game Artificial Intelligence*. Morgan & Claypool Publishers, 2019, vol. 3, no. 2, https://gaigresearch.github.io/gvgaibook/.