Optimising Level Generators for General Video Game AI

Olve Drageset, Mark H.M. Winands
Department of Data Science and Knowledge Engineering
Maastricht University
Maastricht, NL
o.drageset@student.maastrichtuniversity.nl,
m.winands@maastrichtuniversity.nl

Raluca D. Gaina, Diego Perez-Liebana
Game AI Research Group
Queen Mary University of London
London, UK
{r.d.gaina, diego.perez}@qmul.ac.uk

***Abstract*—Procedural Content Generation is an active area of research, with more interest being given recently to methods able to produce interesting content in a general context (without task-specific knowledge). To this extent, we focus on procedural level generators within the General Video Game AI framework (GVGAI). This paper proposes several topics of interest. First, a comparison baseline for GVGAI level generators, which is more flexible and robust than the existing alternatives. Second, a composite fitness evaluation function for levels based on AI playtesting. Third, a new parameterized generator, and a Meta Generator for performing parameter search on such generators are introduced. We compare the Meta Generator against random and constructive generator baselines, using the new fitness function, on 3 GVGAI games: Butterflies, Freeway and The Snowman. The Meta Generator is suggested to perform on par with or better than the baselines, depending on the game. Encouraged by these results, the Meta Generator will be submitted to the 2019 GVGAI Level Generation competition.**

***Index Terms*—GVGAI, level generation, genetic algorithm**

I. Introduction

Procedural Content Generation (PCG), and especially procedural generation of video game levels, has been popular for decades. While its traditions stretch all the way back to the ASCII dungeons of Rogue, PCG has seen a new dawn in combination with different techniques from Artificial Intelligence [1]. In the last years, research has focused on methods that generate content in a general way, in an attempt to reduce the amount of domain knowledge used. The aim of this research is to focus more on the generation algorithms rather than in specific heuristics. A clear example of this is the General Video Game AI (GVGAI) framework. GVGAI is a benchmark that, among other challenges, proposes the investigation of general methods for procedural level generation. Within this context, the level generation track [2] prompts the generator to generate a level for previously unseen games.

The GVGAI framework provides several sample generators for levels, including genetic and constructor generators. A common practice for participants is to tune the parameters, architecture or fitness functions of these sample generators. This paper proposes a new generator that aims at facilitating testing and boosting novel generator architectures by using AI-assisted play testing.

In particular, this paper makes four contributions. Firstly, it proposes a fitness function for evaluating the quality of generated levels, that uses an array of weighted factors. Secondly, it proposes a method for boosting any fast and resource light level generator algorithm: Using this or any other fitness function for selecting the best out of many generated results. Building on these two, it proposes a Meta Generator that optimises the parameters of fast resource light generators, using a fitness function for levels generated to estimate the quality of the generator parameters. Lastly, we propose a fast parametrisable level generator for the Meta Generator to optimise, that builds on the principles of an existing benchmark generator from the GVGAI framework.

The structure of this paper is as follows: Section II gives a brief introduction of the GVGAI framework and the level generation competition track. Section III provides an overview of the related work in this area. Then, Section IV describes the proposed level generators employed in the experiments, which results are discussed in Section V. Finally, conclusions and opportunities for further work are detailed in Section VI.

II. GVGAI

The General Video Game AI (GVGAI) framework is a Java benchmark, evolved from the original Py-VGDL implemented by Tom Schaul [3], which proposes a series of AI challenges

Fig. 1: Human-designed levels for GVGAI games (left: *The Snowman*; top right: *Butterflies*; bottom right: *Freeway*).

978-1-7281-1884-0/19/$31.00 ©2019 IEEE

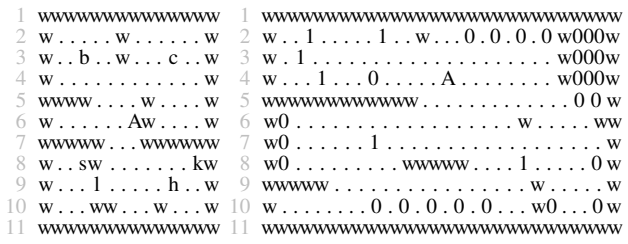

Fig. 2: VGDL levels: *The Snowman* (left), *Butterflies* (right).

around the concept of general game AI. In particular, GVGAI offers a platform for learning, planning and content generation problems in games described in the Video Game Description Language. Each one of the problems have been proposed to the community in the form of competitions [4], [5] and the framework is widely used for research and education [6].

A. Games

The GVGAI framework counts, at the time of writing, more than 180 single and two-player games, with 5 levels each. The games are written in the Video Game Description Language (VGDL), which allows games to be easily created around sprites with certain properties and behaviours and their interactions. These elements are described in VGDL using two files (game and level definition). The game description file is composed of four different *sets*. The *Sprite Set* defines properties for the different entities in the game. The *Interaction Set* describes the effects produced by sprite collisions, such as destroying sprites or causing score changes. The *Termination Set* defines the conditions that lead to an end-game state, determining also if the game is won or lost for the player(s). Finally, the *Level Mapping* specifies the connections between sprites and ascii characters used in the level definition file.

Levels are described in their own ascii files. Each character in a file is mapped to one or more sprites as indicated in the *Level Mapping*. Figure 2 shows examples of VGDL levels representing their initial states. Figure 1 shows the 3 different GVGAI games used in this study. Their variety also shows the complexity and expressiveness that achievable with VGDL:

The Snowman: This is a deterministic game in which the player must push different parts of a snowman (body, trunk and head) into a platform, in the natural body to head order. Parts of the level may be locked that can only be opened if the player collects a key. Points are awarded for correctly placing each part of the snowman.

Butterflies: This is a stochastic game in which the player must capture as many butterflies as possible before all cocoons in the level are opened, in which case the game is lost (cocoons open on contact with butterflies to create new butterflies). When all butterflies are captured, the game is over and won. All butterflies move at random. Points are awarded for each butterfly captured.

Freeway: A version of the original with the same name, this stochastic game puts the player in control of an agent that must cross consecutive roads with incoming traffic. Every time the player gets hit by a vehicle, they lose 1 life (out of 5 available). The player wins if they are able to reach the randomly positioned goal at the other end of the crossings. In contrast to the others, this game has no score, only a victory condition.

B. Game-Playing Agents

The framework exposes an API for planning agents, which includes access to a forward model (FM). This forward model can be used to simulate possible future states of the game when supplied with an action to execute from any state. In the competition setting, each agent has a certain established time to return an action before being disqualified.

GVGAI includes a series of sample controllers to help practitioners in the creation of agents for the benchmarks. Among these controllers, one can find simple ones, like RANDOM (which executes random actions at each time step), DONOTHING, where action NIL (or no-op) is applied at each step, and One Step Lookahead (OSLA). The latter sample controller explores every game state reachable from the current state using the FM, evaluating it with a heuristic that promotes proximity to certain sprites. In this study, DONOTHING and OSLA are 2 of the 3 agents used to evaluate generated levels.

The third controller is YOLOBOT, which is not provided with the framework. Instead, YOLOBOT was a (multiple) competition winner submission developed by Joppen *et al*. [7]. This approach is a combination of two different methods: a heuristic-guided Best First Search used in deterministic games, and a Monte Carlo Tree Search (MCTS; [8]) used for stochastic environments. Used in conjunction with informed priors and rollouts, backtracking and pruning, this agent was able to win three editions of the Single-Player Planning GVGAI competition. The reader is referred to [7] for details.

C. GVGAI Level Generation

The Level Generation track was introduced in 2016 with the objective of proposing a challenge for automatic generation of levels for any game that is given [2]. Participants submit generators that must produce a level for games (unknown a priori) in no more than 5 hours of CPU computation. During this time, the generator can make use of planning agents to play-test potential levels to be returned.

Generators have access to the following game information:

Avatar sprites, controlled by the player.
Solid sprites, static in the game.
Harmful sprites, which kill the player (or can create sprites that would kill the player).
Collectible sprites, which can be picked up by the player.
Other sprites that do not fall in the previous categories.

Additionally, the generator is provided with access to the level mapping, the interaction and termination sets. In return, the generator must provide a 2-dimensional array of characters that forms the level, in the same format as those shown in

Figure 2. The GVGAI Level Generation track was initially proposed in tandem with three distinct level generators [2]:

RANDOM: This generator picks a level size by looking at how many sprite types exist. After surrounding the board in a solid frame, it places at least one of each sprite type on the board. It makes sure to keep around 80% of the space open. If there is additional space left over after these initial constraints are met, the remaining space is filled randomly from the set of sprites. This approach has the advantage that, without looking into the semantics of the sprites, all sprites that are vital for completing the level (goal sprites) are likely to be included in the level. It's also likely that the avatar has space to move.

CONSTRUCTIVE: This generator uses the same level size selection, and includes a frame of static sprites. It also builds connected walls coming out of the frame, while making sure that all open space on the board is connected. This makes for more interesting and deliberate-looking levels for games that rely on labyrinth- and room-like structure. The constructive generator places enemies at a distance from the avatar, to make sure the player doesn't die immediately after spawning. It does not, however, guarantee that all sprites will be used at least once.

GENETIC: This generator initializes a population of levels using the constructive or random generator, before performing evolution using a fitness function. It keeps one population for infeasible levels, and one for feasible levels, that do not mix. The fitness evaluation function is based on two factors, which will be compared to our own proposed fitness function in the methods section.

III. RELATED WORK

Previous entries to the GVGAI Level Generation competition [2] have not been very effective at creating interesting or even playable levels [6]. However, some approaches do succeed in producing quality results. Neufeld et al. [9] use a (+) evolutionary algorithm to evolve the rules used by an Answer Set Programming (ASP) level generator in GVGAI. These levels are then evaluated using a simulation-based method: the fitness of each level is the difference of average scores obtained by vanilla Monte Carlo Tree Search and a random player. Their results showcase the benefits of the concept, although we identify the computational overhead of translating VGDL games into ASP rules as a drawback, especially in the context of the GVGAI competition. This paper uses a similar approach, compatible with the GVGAI competition and applied to a parameterized random generator.

Four out of six submissions to the GVGAI Level Generation competition are based on evolutionary algorithms, using AI simulations for evaluating generated levels [6], although they have not yet been successful in winning the competition. Given previous promising approaches [9], we focus on improving simulation-based evolutionary methods for this task.

A different approach used in the Level Generation competition is using design patterns within various techniques. Sharif et al. [10] analyse the games in GVGAI to identify interesting design patterns, such as solid sprites often forming rooms (almost fully enclosing a section of a level) or collectible sprites often being placed together. Beaupre et al. [11] later use such design pattern analysis to develop general generators which produce levels inspired by the human designs. They use the sample constructive generator provided with the GVGAI framework to generate an initial population of levels, which are evaluated based on the patterns they contain. This population is then evolved to match the pattern weights extracted from the existing GVGAI corpus of games. A final human evaluation of the resultant level shows preference towards the pattern-based levels. However, there is no indication as to the level quality in terms of playablity. We choose to focus on simulation-based evaluations to take into consideration the impact on player game-play, but we consider design patterns additions as a path for further extending the current work.

Several authors explore the use of Relative Algorithm Performance Profiles (RAPP) [12] to evaluate generated games or levels: the difference in performance between proficient and less skilled players is often seen as an indicator of a game's skill depth, with higher skill depth being a desired quality of generated game content. Nielsen et al. [12] compare the relative performance of seven different agents on a set of VGDL games and their results support the correlation between higher-quality games and a larger difference between good and less-skilled players. More recently, Liu et al. [13] use a similar measure to evolve game parameters instead, with similarly good results. Inspired by positive results, we use this same notion in the fitness function and measure the difference in win rate and score between YOLOBOT [7], a high-performing bot in the GVGAI planning competition, and two simple agents, One-Step Look Ahead (OSLA) and DO NOTHING. See Section II-B for details of agents used in this work.

One of the aspects we consider in this paper is the importance of using the right parameters for the generator. Manuel et al. [14] evolve level generators for Super Mario Bros interactively, with human supervision. They use both a measure of the playablity of generated levels (using simulation-based evaluation) as well as the human preference input in order to evolve better levels. We take a similar approach, while excluding the human factor in order for our method to be compatible with the GVGAI competition and entirely autonomous, while also testing our Meta Generator's performance on several games. While most generator optimizers use fairly simple evolutionary algorithms, Lucas et al. [15] propose a model-based approach for tuning game parameters. Both GVGAI games and agents can be stochastic, which introduces considerable noise in the evaluation. Additionally, simulation-based evaluations are expensive, as (potentially multiple) games have to be run in order to test the quality of the generated solutions. The N-Tuple Bandit Evolutionary Algorithm (NTBEA) is shown to perform well in noisy environments as well as being sample efficient [16], which could lead to better results within the short timespan allowed in the GVGAI competition. NTBEA has further been shown to produce good results when optimising player experience (represented by score curves) in

GVGAI games [17], thus we consider this as the next step for improving our method further.

IV. Level Generators

This section describes the methods used in our experiments, and details our contributions: The Population Generator, the parameterized Percentage-Wise Generator (PWG), the Meta Generator, and the fitness evaluation function.

A. Population Generator Baselines

One of the problems when building level generators for the GVGAI Level Generation competition is that after the generator is built and the fitness evaluation function is designed, there is no good baseline measure to compare the generator's output quality with. Out of the baseline generators given by the competition organizers, while the GA relies on a fitness function and validates the playability of all levels before returning them, the random and constructive generators produce a single level without playtesting or validating it. From this point on we will refer to the latter approach as `one-shot generation`. Our hypothesis is that, even without performing any genetic operations, testing and validating levels is a large part of the performance difference between the genetic algorithm and the one-shot generators. This motivated us to build a framework around continuous one-shot generators (called the Population Generator), as well as a fitness function for noisy evaluation of generated levels. This turns a one-shot generator into a continuously improving generator that can keep generating and testing levels for any amount of time. When time runs out, it returns the best level in the population, according to the fitness function.Fitness values are normalized across the entire population of levels generated for a game.

B. Percentage-Wise Generator

The Percentage-Wise Generator (PWG) is the base generator used in our experiments. The PWG was designed with a desire to minimize the assumptions we make about the game we are generating levels for, and to increase generality by minimizing the reliance on human-injected bias. The sample constructive generator in the GVGAI Level Generation competition is a counter example. Because of these considerations, the PWG is based loosely on the random level generator. The biggest difference is that it is parameterised, taking into account the following when generating levels:

Percentage of each sprite that should be used.
Whether the $(x; y)$ coordinates of each sprite should be sampled from a Gaussian distribution.
Mean of the distribution.
Standard deviation of the distribution.
Size of the level and whether a border of static wall should be placed around the level.

Details of the parameter search space are depicted in Table I (float parameters are continuous, bounded between 0 and 1). The Mean and St.dev parameters are scaled by the Width and Height parameters. The PWG only uses information about three types of sprites: the avatar, walls, and open space. In terms of injected bias, the PWG knows that exactly one avatar must exist, that a frame of static walls is an option (enabling the frame is a boolean parameter), and that having more than half of the area of the level being covered by open space is a good place to start the optimisation process. It is not, however, told anything about the purpose or function of any of these sprites or their interactions. The GVGAI Level Generation competition explicitly gives access to this knowledge, and several other generators use it. We chose to avoid using it in order to make our method as general and requiring as little information as possible.

TABLE I: Generator parameter search space

Parameter	Type	Search space
Sprite usage$_n$	Float	[0,1]
x-Gaussian sampling$_n$	Boolean	true, false
y-Gaussian sampling$_n$	Boolean	true, false
x-Distribution mean$_n$	Float	[0,1]
y-Distribution mean$_n$	Float	[0,1]
x-Distribution st.dev$_n$	Float	[0,1]
y-Distribution st.dev$_n$	Float	[0,1]
Level border	Boolean	true, false
Width	Integer	[4,18]
Height	Integer	[4,18]

C. Meta Generator

The Meta Generator relies on optimising the parameters of other level generators (in our case, the PWG). The fitness evaluation of the sub-generators involves generating and evaluating levels using the fitness function described in Section IV-D. By addressing the generator optimisation task, the level generation task is implicitly addressed as well.

1) Motivation: The level search space for GVGAI games is large, and current methods for simulation-based level fitness evaluation are relatively expensive. Thus we hypothesize that genetic level generators rely on having a good population initialization in order to find a "good" (according to a fitness function) solution within the allocated time. It would therefore be optimal to have a much faster generator that delivers many good levels, before using a genetic algorithm to improve upon them to find one high-quality level. Having a generator that can quickly and consistently make levels that are even close to being good for any never-seen-before games is difficult.

The idea behind the Meta Generator is to optimise the parameters of a fast generator. The best levels generated during the search can be used as the initialization set for genetic search in the level space, or the best level found so far can be returned directly. Additionally, this system can be used to supply a player with a continuous stream of levels, by running the optimized Meta Generator in the background during play.

2) Generator Population and Level Population: When optimising the parameters for the PWG, we maintain a population of generator parameters. This population keeps track of which generators produced what levels, so that we can calculate the fitness of each generator as the average fitness of the levels it has produced. A combined population of all the levels generated and evaluated so far by the Meta Generator is also maintained. Just like for the Population Generators, the

information from all the levels generated is used to calculate the fitness of each individual level, normalized across the population of levels generated for a game.

3) Parameter Optimisation Algorithm: The Meta Generator uses mutation and crossover to search the parameter space of the PWG. It uses Upper Confidence Bounds (UCB; [18]) to select in which direction to guide its search. The UCB formula is depicted in Equation 1, split into 2 terms for exploitation (first) and exploration (second). V_i is the value estimation of the generator i. This value is the average fitness of the levels generated so far, and the assumption is that it translates to the generator being a good candidate for crossover and mutation. C is the exploration parameter, which is set to $\sqrt{2}$. N is how many levels have been generated in total, and n_i is how many levels have been generated by the generator.

$$UCB = V_i + C\sqrt{\frac{\ln N}{n_i}} \quad (1)$$

The UCB value balances exploration of areas that have not been explored and the exploitation of areas that are promising. As outlined in Algorithm 1, the Meta Generator focuses its search on the generator with the highest UCB value, using a (1,) roulette wheel (fitness proportionate) selection strategy. This means the generator with the highest UCB value is picked, and other generators are selected and crossed with the top generator. Each time a generator is selected for reproduction, one more level is generated using its parameters before continuing. This encourages the noisy fitness evaluation of the more promising generator parameters to become more accurate, which means we avoid repeatedly using parameters for crossover that are objectively unfit, but inaccurately evaluated. In the case where all results of crossovers and mutations are worse than the previous population, forcing the existing generators to keep producing levels further ensures that we keep exploiting our current best generators.

Algorithm 1 Meta Generator pseudo code. It optimizes level generator parameters, returns the level with the highest fitness.

Input: Parameterised one-shot generator g
Input: Fitness function f
Input: Time budget t
Output: A GVGAI Level.

```
1:  gPop ← initializePopulation(g)
2:  lPop ← Empty
3:  while time < t do
4:      generator ← gPop:maxUCB()
5:      level ← generator:generateLevel()
6:      fitness ← f(level)
7:      lPop:add(level; fitness)
8:      gPop:update(g; level)
9:      parents ← rouletteSelect( ; gPop)
10:     offspring ← crossover(g; parents)
11:     offspring:mutate()
12:     for child in offspring do
13:         levels ← child:generate(n)
14:         fitnesses ← f(levels)
15:         lPop:add(levels; fitnesses)
16:         gPop:add(child; levels)
17:     end for
18: end while
19: level ← lPop:best()
20: return level
```

D. Noisy Level Fitness Evaluation

The fitness evaluation function for levels is a compound measure of 8 factors, that are calculated from the data of several General Video Game AI agents playing the level (see Section II-B for details of agents used).

f_1 WIN FACTOR: This factor measures how much better YOLOBOT performs than OSLA and DO NOTHING, in terms of win percentage (every game in GVGAI can be won or lost).

f_2 SCORE FACTOR: This factor measures how much better YOLOBOT performs than OSLA and DO NOTHING in terms of game score. Looking at the difference between the performance of agents with various skill levels as an indicator of game skill depth has been tried before in adversarial games [13] and single player games [12].

f_3 DANGER FACTOR: This factor measures how close the AI is to death on average throughout the game. The danger score at a given frame is calculated by doing m random roll-outs of length n, and seeing how many of them end up in death. The DANGER FACTOR is the average danger score for the round. While game AI evaluations have been used before to estimate the difficulty of games [19], danger should not be confused with difficulty. An agent can have a 100% win rate, and still experience a high sense of danger throughout the game. Conversely, it can also experience no danger at all, up until the point at which it loses the game.

f_4 DANGER RATE FACTOR: This factor measures the presence of danger rather than the degree of it. It returns the percentage of time the AI is within reach of death, i.e. any one of the danger-measuring roll-outs at a game tick ends in death.

f_5 INTERACTION FACTOR: This factor measures how many interactions take place between the avatar, its spawned attacks (missiles, sword strikes), and the other sprites in the level. The assumption is that having more interactions is more interesting.

f_6 UNIQUE INTERACTION FACTOR: It measures how many unique types of interactions the avatar and its spawned attacks have with the environment. The assumption is that a player is more stimulated by a level that encourages more of these interaction types to take place.

f_7 LENGTH FACTOR: If a level is solvable, it is desirable that the solution takes longer. The assumption is that a longer solution is less likely to feel trivial to the player.

f_8 SOLVABILITY FACTOR: It indicates if any of the three

AI agents could solve the level in any of their attempts.

All factors (except f_8) are measured over a number of simulations, and because the agent and the games are stochastic, results may vary. The total fitness score of a level is calculated according to Equation 2 (solvability is excluded), where w_n is the weight of the n-th factor. Each individual factor is normalized in the range [0,1], according to all other observed values for that factor for the same game. The weights should preferably be adjusted in accordance with human preference, but, for the lack of such data, they were set, from w_1 to w_7, as follows: (3,2,2,1,1,1,1).

$$f_L = \frac{\sum_{n=1}^{7} f_n * w_n}{\sum_{n=1}^{7} w_n} \quad (2)$$

This puts fitness in the range $[0; 1]$. If the level is unsolvable, the fitness becomes $f_L - 1$, putting it in the range $[-1; 0]$. This is done to reflect that any solvable level is better than an unsolvable level: the player should be able to win.

V. Experiments

The goal of the experiments is to compare the performance of the random and constructive generators against the Meta Generator proposed in this paper, when provided with a 5 hour budget as in the GVGAI Level Generation competition. Each of these 3 generators was set to generate levels for 3 Eh is repeated 5 times. The experiments were performed on IBM System X iDataPlex dx360 M3 Server nodes, where each had one Intel Xeon E5645 processor core allocated to it, and a maximum of 2GB of RAM of JVM Heap Memory.

A. Results

Figure 3 shows the average fitness (over 5 runs) of the best level generated so far (the one that would be returned at that point), by each of the generators. The difference between the performance of the three generators depends on the game.

1) Butterflies: Butterflies is a simple game in which a random spread of sprites can lead to a level that is challenging and plays well. It therefore makes sense that the Meta Generator only makes marginal gains on the baselines. When we analyse the levels generated by each of the generators, it becomes apparent that the small difference in fitness actually accounts for the fact that the Meta Generator is able to build levels of much larger size. The random and constructive generators restrict the size of their levels because of the small number of sprite types. A larger level makes for longer games, and due to the Length Factor (f_7) this adds to the fitness on solvable levels. We see the larger levels produced by the Meta Generator as more enjoyable for humans than the very compact ones produced by the baselines.

2) Freeway: Freeway is a more complicated game. In addition to also being stochastic, it has many more sprite types. In addition to this, traditional *Freeway* levels are all structured in a specific way: the car spawners are located in key places in order for the game to be recognizable and make sense to humans. While random scattering of sprites works well for

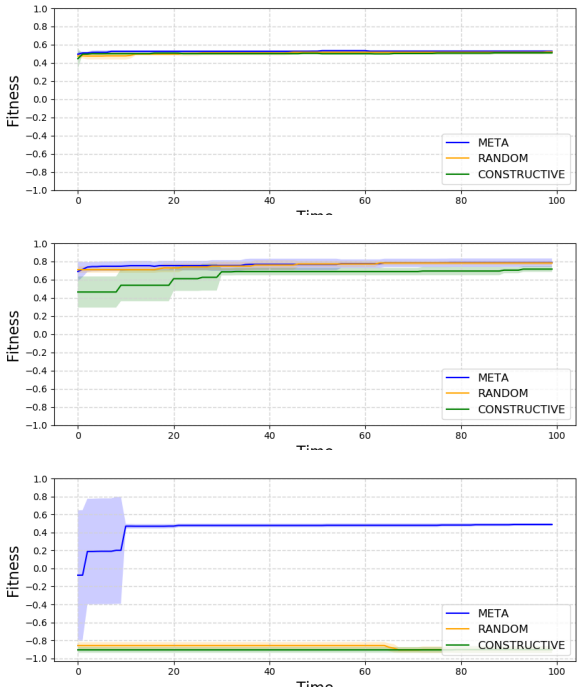

Fig. 3: The average fitness of the best level of each generator over the course of 5 hours of continuous generation. From top to bottom: *Butterflies*, *Freeway* and *The Snowman*. Colored area indicates standard deviations.

Butterflies, it does not for *Freeway*. The constructive generator uses and builds in a specific way that is good for dungeon-style maps, but it is not effective in *Freeway* as we can see in Figure 3 *b*). From the lower middle fitness distribution in Figure 5 we can see that the constructive generator produces a considerable amount of low fitness (but playable) levels, as well as some levels on par with the other generators. Interestingly, when inspecting the final generation of *Freeway* levels, the constructive generator is the only one that managed to construct a level that is somewhat sensible (more on this later). The Meta Generator and its PWGs have no way of biasing the levels produced to have patterned structure and spacing the sprites in a certain way, which is what we imagine is most beneficial for *Freeway*.

3) The Snowman: The Snowman is the only puzzle game out of the three. It is also the game where the largest difference in fitness is observed. The Meta Generator outperforms the constructive and random generators, who do not generate any levels that are solvable by YOLOBOT. While the Meta Generator starts out poorly as well, it explores different generator parameter until it finds a playable level, and starts using the parameters of the successful PWG as a basis for further exploration. As can be observed in the right-most row of histograms in Figure 5, this leads to not only one or two playable levels, but a sizable population of playable levels. Upon further inspection, it turns out that the levels generated by the constructive and random generators are actually fairly

simple to solve for humans, but they are consistently too large and too cluttered for YOLOBOT to solve. The Meta Generator is the only one out of the three that manages to create smaller, relatively uncluttered levels that YOLOBOT can solve, because it is free to change the size of the level and cover percentage of all the sprite types.

4) Summing up: The constructive generator brings a stronger bias into its levels, and that seems to disadvantage it slightly going up against the less biased random and Meta generators. The random generator also has a strong bias in size and cover percentages that disadvantages it in a situation like this, where there is an opportunity to sample a large amount of different combinations and test their fruitfulness. The Meta Generator achieves a similar performance to random, although outperforming both baselines in *The Snowman*.

B. Generated Levels

Observe in Figure 4 examples of levels returned at the end of 5 hours of generation, by each of the generators. For *Butterflies*, we can see how the size restrictions imposed by the random and constructive generators limit them from discovering the benefit of a larger level. For *Freeway*, we can see that the basic idea of the game is completely deconstructed, as the floor tiles where the goal and player can spawn (light grey) are spread out completely, and are dangerously placed. In *Freeway* levels, YOLOBOT is superhuman at avoiding fast moving danger, so implementing a more human-like behaviour with longer reaction time similar to [2] might help bias the levels to be less hectic. The constructive generator (right) surprised by creating one level where 3 out of 4 spawn points are safe from traffic, but this is purely by chance. While the levels generated by the Meta generator on the two other games differ visually from the Population Generators, the similarity between the Random and Meta on Freeway is striking. For *The Snowman*, we observed that levels returned by the random and constructive generators were generally cluttered and large. The size is determined by the sprite set, and the percentage of the board that is to be filled with sprites is not variable. The Meta Generator tended to return sparser and smaller levels.

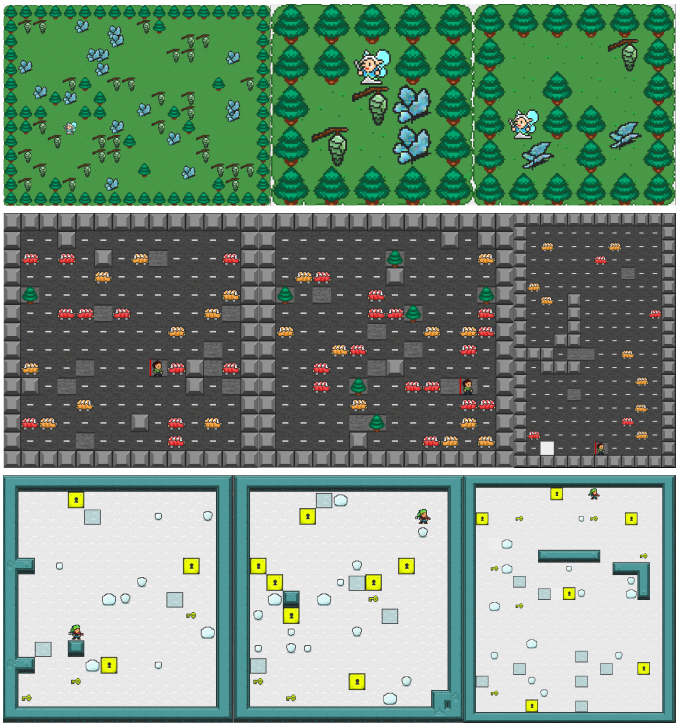

Fig. 4: A sample of the best levels returned at the end of 5 hours. From top to bottom: *Butterflies*, *Freeway* and *The Snowman*. From left to right: Meta, Random, Constructive.

VI. Conclusions and Future Work

In this paper we describe a fitness evaluation function based on several factors, which is used by 3 different generators to evaluate the quality of generated levels in the General Video Game AI framework (GVGAI): random, constructive and our proposed Meta Generator. The Meta Generator builds upon a parameterized version of the random generator and evolves its parameters in order to produce better levels. The random and constructive generators are tested in continuous runs over 5 hours (as per the GVGAI Level Generation competition rules) in 3 GVGAI games, *Butterflies*, *Freeway* and *The Snowman* and are shown to perform similarly or worse than the Meta Generator (using the same budget), depending on the game.

The constructive generator brings a strong bias into its levels, which seems to disadvantage it going up against the Random and Meta generators. The Meta Generator gains in fitness over both the Population Generators from its extra flexibility, and this flexibility seems to have a larger impact in games that are more difficult to produce levels for.

The next experiment lined up is to compare the performance of the Meta Generator against a strong genetic level generator, such as the original GVGAI genetic baseline [2], using comparable one-shot generators for population initialization. This Meta Generator will be entered into the 2019 GVGAI Level Generation competition, where it can be tested rigorously against other generators. A detailed comparison with previous competition entries is also considered for future work.

The parameter optimisation performed by the Meta Generator could be further improved. A more powerful method such as The N-Tuple Bandit Algorithm (NTBEA) [15] [20] could be used, which has been shown to work well for online parameter tuning [21]. One line of work would be using a population of NTBEAs larger than the thread pool. After a thread completes a step on one NTBEA instance, the thread picks which Meta Generator instance to work on next by using UCB to select from those available.

The weighting of the factors in the current fitness function could also be adjusted so that the fitness score aligns better with human experience. Browne and Maire [22] focused on human experience in their approach, which contributed to the creation of the award-winning board game Yavalath. It can be applied similarly to the evaluation of video game levels, by extracting features of human play together with their explicit preference indications for a level.

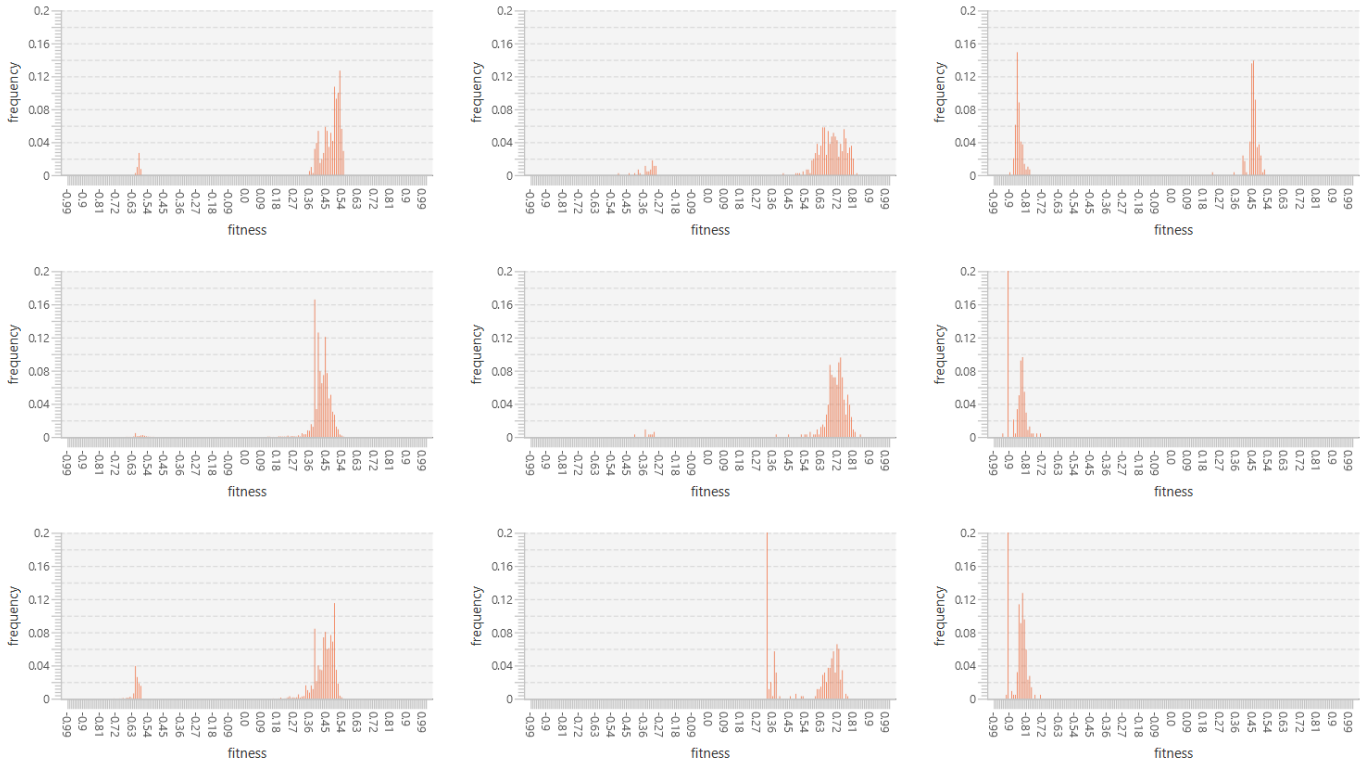

Fig. 5: The fitness distributions of all generators on all games. Each row contains one generator, each column one game. From top to bottom: Meta, Random, Constructive. From left to right: *Butterflies*, *Freeway*, *The Snowman*.

ACKNOWLEDGMENT

This work was partially funded by the EPSRC CDT in Intelligent Games and Game Intelligence (IGGI) EP/L015846/1.

REFERENCES

[1] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.

[2] Ahmed Khalifa, Diego Perez-Liebana, Simon M Lucas, and Julian Togelius. General Video Game Level Generation. In *Proc. of the 2016 Annual Genetic and Evolutionary Comp. Conf.*, pages 253–259, 2016.

[3] Tom Schaul. A Video Game Description Language for Model-based or Interactive Learning. In *Proceedings of the IEEE Conference on Computational Intelligence in Games*, pages 193–200, 2013.

[4] Diego Perez-Liebana et al. The 2014 General Video Game Playing Competition. *IEEE Tran. on CI and AI in Games*, 8(3):229–243, 2016.

[5] Raluca D. Gaina et al. The 2016 Two-Player GVGAI Competition. *IEEE Transactions on CI and AI in Games*, 2017.

[6] Diego Pérez-Liébana et al. General Video Game AI: a Multi-Track Framework for Evaluating Agents, Games and Content Generation Algorithms. *CoRR*, abs/1802.10363, 2018.

[7] Tobias Joppen, Miriam Moneke, Nils Schröder, Christian Wirth, and Johannes Fürnkranz. Informed Hybrid Game Tree Search for General Video Game Playing. *IEEE Trans. on Games*, 10(1):78–90, 2018.

[8] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4:1:1–43, 2012.

[9] Xenija Neufeld, Sanaz Mostaghim, and Diego Perez-Liebana. Procedural Level Generation with Answer Set Programming for General Video Game Playing. In *7th Computer Science and Electronic Engineering (CEEC)*, pages 207–212. IEEE, 2015.

[10] Mudassar Sharif, Adeel Zafar, and Uzair Muhammad. Design Patterns and General Video Game Level Generation. *Intl. Journal of Advanced Computer Science and Applications*, 8(9):393–398, 2017.

[11] Spencer Beaupre, Thomas Wiles, Sean Briggs, and Gillian Smith. A Design Pattern Approach for Multi-Game Level Generation. In *AI and Interactive Digital Entertainment*, pages 145–151. AAAI, 2018.

[12] Thorbjrn S. Nielsen et al. General Video Game Evaluation Using Relative Algorithm Performance Profiles. In *In Applications of Evolutionary Computation*, pages 369–380. Springer, 2015.

[13] Jialin Liu et al. Evolving Game Skill-Depth Using General Video Game AI Agents. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 2299–2307, 2017.

[14] Manuel Kerssemakers, Jeppe Tuxen, Julian Togelius, and Georgios N Yannakakis. A Procedural Procedural Level Generator Generator. In *IEEE Conf. on Comp. Intelligence and Games*, pages 335–341, 2012.

[15] K. Kunanusont, R. D. Gaina, J. Liu, D. Perez-Liebana, and S. M. Lucas. The N-Tuple bandit evolutionary algorithm for automatic game improvement. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 2201–2208, June 2017.

[16] Simon M Lucas et al. Efficient Evolutionary Methods for Game Agent Optimisation: Model-Based is Best. *arXiv:1901.00723*, 2019.

[17] Kamolwan Kunanusont, Simon M Lucas, and Diego Perez-Liebana. Modelling Player Experience with the N-Tuple Bandit Evolutionary Algorithm. In *AI and Interactive Digital Entertainment*, 2018.

[18] Peter Auer, Nicolò Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2):235–256, 2002.

[19] Hendrik Horn, Vanessa Volz, Diego Pérez-Liébana, and Mike Preuss. MCTS/EA Hybrid GVGAI Players and Game Difficulty Estimation. In *Conf. on Computational Intelligence and Games*, pages 1–8. IEEE, 2016.

[20] S. M. Lucas, J. Liu, and D. Perez-Liebana. The N-Tuple Bandit Evolutionary Algorithm for Game Agent Optimisation. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–9, July 2018.

[21] C. F. Sironi, J. Liu, and M. H. M. Winands. Self-Adaptive Monte-Carlo Tree Search in General Game Playing. *IEEE Tran. on Games*, 2018.

[22] C. Browne and F. Maire. Evolutionary Game Design. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(1):1–16, March 2010.