Chapter 6 - Procedural Content Generation in GVGAI

Ahmed Khalifa and Julian Togelius

Procedural Content Generation (PCG) is to use a computer program/algorithm to generate game content [16] automatically. This content could be anything in the game such as textures [20], levels [12,8], rules [7], etc. PCG might be a little new in research but it has been around since the beginning of computer games. It was used as a game design element to allow replayability and new scenarios such as in Rogue (Glenn Wichman, 1980). Also, it was used to allow new types of games with huge amount of content using a small foot print such as in Elite (David Braben and Ian Bell, 1984). Although more and more people have been adapting and using PCG in their games and discovering new techniques, no one was tackling the problem of generality. Most of the used techniques in the industry are all constructive [15] and depend on a lot of hacks and tricks based on the game knowledge. Algorithms that can adapt and work between different games are still a dream for the PCG field. That is the main motivation behind having a PCG track in the GVGAI. We wanted to give the people a framework to help them research towards finding more generic ways that can generate new content with small amount of information about the current game.

GVGAI framework has several different facets that define games: levels, interaction sets, termination conditions, and sprite sets (graphical representation and types). We decided to start with tackling the level generation problem first as level generation is one of the oldest and challenging problems that people are tackling since the early 1980s. Later, we designed a new track for rule generation where the user has to generate both interaction sets and termination conditions given the rest of the facets. One of the core challenges for both tracks is how to define a good way to compare the different generation techniques. For the competition, we relied on humans to provide us with this data based by comparing two generate content from different generators. The humans do the comparisons several times on examples of each generator from different games, to make sure that the algorithm can adapt between different games and also can generate different levels for the same game. This technique works fine for the competition, but it doesn't help the users to understand how to enhance their generator, as there is no specific set of rules that define what a good level or game is. In this chapter, we will talk about these two different tracks: level (Section 1) and rule generation (Section 2). For each one of them, we will dis-

cuss their interface, the sample generators provided, and the latest techniques and generators used.

1   Level Generation in GVGAI

The level generation track was introduced in 2016 as a new competition track [8]. It is considered the first content generation track for the GVGAI framework. The competition focuses on creating playable levels providing the game description. In this track, every participant submits a level generator that produces a level in a fixed amount of time. The framework provides the generator with the game sprites, interaction set, termination conditions, and level mapping, in return, the generator produces a 2D matrix of characters, where each character represents the game sprites at that location. The framework also allows the generator to provide their own level mapping if needed.

```
public abstract class AbstractLevelGenerator {

    public abstract String generateLevel(GameDescription game, ElapsedCpuTimer elapsedTimer);

    public HashMap<Character, ArrayList<String>> getLevelMapping()
    {
        return null;
    }
}
```

Fig. 1: AbstractLevelGenerator Class functions.

The game information is provided to the generator using a `GameDescription` object and `ElpasedCpuTime` object. Figure 1 shows the `AbstractLevelGenerator` class that the participants should extend to create their level generator. A `GameDescription` object is a structured data that provide access to the game information in a more organized manner, while the `ElpasedCpuTime` object provides the user with the remaining time for the generator to finish. When running the competition, the `ElpasedCpuTime` timer gives each generator five hours to finish its job.

```
1  BasicGame
2    SpriteSet
3      floor      > Immovable randomtiling=0.9 img=oryx/floor3 hidden=True
4      goal       > Door img=oryx/doorclosed1
5      key        > Immovable img=oryx/key2
6      sword      > OrientedFlicker limit=5 singleton=True img=oryx/slash1
7      movable >
8        avatar   > ShootAvatar stype=sword frameRate=8
9          nokey            > img=oryx/swordman1
10          withkey          > img=oryx/swordmankey1
11        enemy      > RandomNPC
12          monsterQuick  > cooldown=2 cons=6   img=oryx/bat1
13          monsterNormal > cooldown=4 cons=8   img=oryx/spider2
14          monsterSlow   > cooldown=8 cons=12 img=oryx/scorpion1
15      wall       > Immovable autotiling=true img=oryx/wall3
16
17    InteractionSet
18      movable  wall     > stepBack
19      nokey    goal     > stepBack
20      goal     withkey > killSprite  scoreChange=1
21      enemy    sword    > killSprite  scoreChange=2
22      enemy    enemy    > stepBack
23      avatar   enemy    > killSprite  scoreChange=-1
24      nokey    key      > transformTo stype=withkey scoreChange=1 killSecond=
             True
25
26    TerminationSet
27      SpriteCounter stype=goal    win=True
28      SpriteCounter stype=avatar win=False
```

Listing 1: VGDL Definition of the game *Zelda*.

Beside this object, the framework provides a helper class called `GameAnalyzer` which analyzes the `GameDescription` object and provide additional information that can be used by the generator. The `GameAnalyzer` divides the game sprites into five different types.

- Avatar Sprites: sprites that are controlled by the player.
- Solid Sprites: static sprites that prevent the player movement and have no other interactions.
- Harmful Sprites: sprites that can kill the player or spawn another sprite that can kill the player.
- Collectible sprites: sprites that the player can destroy upon collision with them, providing score for the player.
- Other sprites: any other sprites that doesn't fall in the previous categories.

The `GameAnalyzer` also provides two additional arrays for spawned sprites and goal sprites. It also provides a priority value for each sprite based on the number of times

Sprite Name	Sprite Image	Sprite Type	Spawned Sprite	Goal Sprite	Priority Value
floor		Other	FALSE	FALSE	0
goal		Other	FALSE	TRUE	3
key		Collectible	FALSE	FALSE	1
sword		Other	TRUE	FALSE	1
nokey		Avatar	FALSE	TRUE	4
withkey		Avatar	FALSE	TRUE	2
monsterQuick		Harmful	FALSE	FALSE	4
monsterNormal		Harmful	FALSE	FALSE	4
monsterSlow		Harmful	FALSE	FALSE	4
wall		Solid	FALSE	FALSE	1

Table 1: The *GameAnalyzer* data for the game of *Zelda*.

each sprite appears either in interactions or termination conditions. The spawned sprites array contains all the sprites that can be generated from another sprite, while the goal sprites array contains all the sprites that appear in the termination conditions of the game. Table 1 shows the `GameAnalyzer` output of the VGDL game *Zelda* defined in Listing 1. `nokey`, `withkey`, and `goal` are the only goal sprites as they appeared in the termination conditions. `sword` is the only spawned sprite in that game as it only appears when the avatar attacks (avatar spawns a *sword* to attack). The sprite types are assigned based on the interactions of these objects with each other and their sprite class. For example: `withkey` and `nokey` are avatar sprites because they are of sprite class `ShootAvatar`, while `monsterQuick`, `monsterNormal`, and `monsterSlow` are harmful sprites as they can kill the avatar sprites upon collision, as defined in the interaction set.

In the following subsections, we describe the sample generators that are provided with the GVGAI framework and competition and other generators that were submitted to the competition in the IJCAI 2016, CIG 2017, and GECCO 2018 or have been published on or before 2018.

1.1   Sample Generators

This section discusses the three sample generators provided with the GVGAI framework in detail. Besides discussing how they work and function, we also elaborate at the end about their advantages over each other and show a previous user study that validates our claims.

Sample Random Generator This is the simplest known generator in the GVGAI framework. The algorithm starts by creating a random sized level directly proportional to the number of game sprite types. Then, it adds a solid border around the level to make sure that all the game sprites will stay inside. Finally, it goes over every tile with 10% chance to add a random selected sprite and makes sure that each game sprite appears at least once and there is only one avatar in the level.

Sample Constructive Generator This is a simple generator that is provided with the framework that uses the `GameAnalyzer` data to improve the quality of the generated levels. Figure 2 summarizes the core steps of the sample constructive generator. The generator consists of four steps, plus pre-processing and post-processing.

1. **(Pre-processing) Calculate cover percentages:** This pre-processing step helps the generator to define the map size and the percentage of tiles that will be covered with sprites. The size of the map depends on the number of game sprites in the game, while the cover percentage is directly proportional with the priority value for each sprite.
2. **Build the level layout:** This is the first step in the generation. The algorithm first builds a solid border around the level then it adds more solid objects inside the level that are connected to each other and not blocking any area.
3. **Add an avatar sprite:** The generator adds a single avatar sprite in any random empty space in the level.
4. **Add harmful sprites:** The generator adds harmful sprites to the map proportional to the distance to the avatar to make sure the player doesn't die as soon as the game starts.
5. **Add collectible and other sprites:** The generator adds collectible and other sprites at random empty locations of the map.
6. **(Post-processing) Fix goal sprites:** The generator makes sure that the number of goal sprites is greater than the specified number of sprites in the termination condition. If this is not the case, the generator adds more goal sprites till this happens.
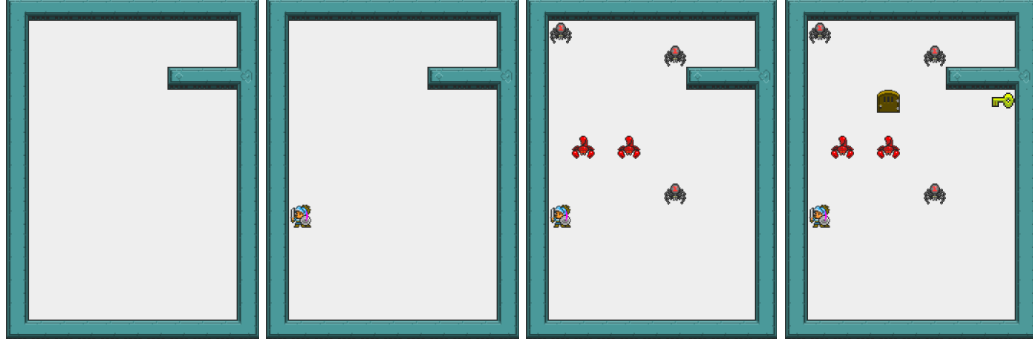
Fig. 2: Steps applied in the constructive generator for VGDL *Zelda*. Top left: Build the level layout. Top right: Add an avatar sprite. Bottom left: Add harmful sprites. Bottom right: Add collectible and other sprites.

Sample Genetic Generator This is a search-based level generator based on using Feasible Infeasible 2-Population genetic algorithm (FI2Pop) [9]. FI2Pop is a genetic algorithm that keeps track of two populations: feasible and infeasible populations. The infeasible population contains the chromosomes that do not satisfy the problem constraints. The feasible population on the other hand tries to improve the overall fitness for the problem. At any point during the generation, if the chromosome doesn't satisfy the problem constraints it gets transferred to the infeasible population and vice versa. The initial population is populated using the constructive generator where each chromosome represents a generated level in the form of 2D array of tiles. The generator uses one point crossover around any random tile and three mutation operators.

- **Create:** create a random sprite at an empty tile position.
- **Destroy:** clear all the sprites from a random tile position.
- **Swap:** swap two random tiles in the level.

We use three different game controllers to evaluate the generated levels: a modified version of OLETS, *OneStepLookAhead*, and *DoNothing*. OLETS is the winner of the 2014 single-player planning track of the GVGAI competition [14]. The algorithm is based on Open Loop Expectimax algorithm (for more details check Chapter 4). The algorithm was modified to play more like a human by introducing action repetition and NIL actions whenever the agent changes direction. This modification was added to influence the generated levels not to require super human reaction time to solve them. *OneStepLookAhead* is just a simple greedy algorithm that picks the best action for the immediate following move. *DoNothing* is a simple algorithm that does not execute any action.

These agents are used as part of the fitness calculation and constraint solving. The feasible population fitness is divided into two parts.

– **Relative Algorithm Performance:** The first part of the fitness calculate the difference in performance between the modified OLETS and *OneStepLookAhead*. This is based on Neilsen et al. [13], which assumes that a good designed level/game has a high performance difference between a good playing algorithm and bad one playing at it.

– **Unique Interactions:** The second part of the fitness calculates the number of unique interactions that fire in the game due to the player or an object spawned by the player. We hypothesise that a good generated level should have enough interactions that can be fired by the player to keep the game interesting.

For the infeasible population, the algorithm tries to satisfy seven different constraints. These constraints were designed based on our understanding of GVGAI and our intuition about a good level.

– **Avatar Number:** each level has to have only one avatar, no more and no less. This constraint makes sure that there is one controllable avatar when the game starts.

– **Sprite Number:** each level has to have at least one sprite from each non spawned sprites. This constraint makes sure that the level is not missing any essential sprite that might be needed to win.

– **Goal Number:** each level has to have an amount of goal sprites higher than the number specified in the termination condition. This constraint makes sure the level does not automatically end in a victory when the game starts.

– **Cover Percentage:** tiles should only cover between 5% and 30% of each level. This constraint make sure the level is neither empty nor crowded with different sprites.

– **Solution Length:** levels must not be solved in less than 200 steps to make sure the level is not trivial.

– **Win:** levels must be winnable by the modified OLETS to make sure that players can win it too.

– **Death:** the avatar must not die in the 40 steps when using the *DoNothing* player to make sure the player does not die in the first couple of seconds of the game.

Pilot Study and Discussion In a previous work [8], the sample generators were compared to each other through a pilot study using 25 human players. The study was conducted on three different VGDL games.

- *Frogs*: is a VGDL port for Frogger (Konami, 1981). The aim of the game is to help the frog to reach the goal without getting hit by a car or drown in water.
- *PacMan*: is a VGDL port for PacMan (Namco, 1980). The aim of the game is to collect all the pellets on the screen without getting caught by the chasing ghosts.
- *Zelda*: is a VGDL port for the dungeon system in The Legend of Zelda (Nintendo, 1986). The aim of the game is to collect a key and reach a door without getting killed by enemies. The player can kill these enemies using their sword for extra points.

For this pilot study, we generated three levels per game by giving each algorithm 5 hours maximum for generation. Each user is faced with two generated levels that are selected randomly and they have to decided if the first level is better than the second, the second level better than the first, both are equally good, or both are equally bad.

	Preferred	Non-Preferred	Total	Binomial p-value
Search-Based vs Constructive	23	12	35	0.0447
Search-Based vs Random	21	10	31	0.0354
Constructive vs Random	17	24	41	0.8945

Table 2: Player preferences for each generator aggregated over the three games.

We hypothesise that the search based levels are better than constructive levels which are better than random generated levels. Table 2 shows the result of that study over all the three games. In that table, we only kept the results when any of the two levels is better than the other, removing all the data where both levels are equally good or equally bad. From the table we can be sure that the search based generated levels are better than both constructive and random but it was surprising to find that constructive generator and random generator were on the same tier. From further analysis of the result, we think the result is the constructive generator didn't make sure there is at least one object from every different sprite type which caused some of the constructive levels to be unsolvable compared to the random generated levels.

1.2 Competition and Other Generators

We can divide the level generator into three main categories based on their core technique: constructive methods, search-based methods, and constraint-based methods. The sample random generator and sample constructive generator are both constructive methods while sample genetic generator is a search-based method. In the

following parts, we are going to talk more about these algorithm and their core technique.

Constructive methods Constructive methods uses generate levels directly by placing game sprites in the level based on game specific knowledge such as don't place player too close to enemies, don't add solid sprites that isolates areas in the map, etc. These generators don't check for playability after generation as they are supposed to be designed in a way to avoid these problems and make sure the generated content is playable all the time.

Easablade constructive generator: is the winner of IJCAI 2016 level generation competition. This generator uses cellular automata [6] on multiple layers to generate the level. The first layer is responsible to design the map layout, followed by the exit sprite and the avatar sprite, then the goal sprites, harmful sprites, and others.

N-Gram constructive generator: was submitted to CIG 2017 level generation competition. It uses a n-gram model to generate the level. The generator uses a recorded playthrough to generate the levels. The algorithm has cases for each different type of interactions, for example: attacking involves adding an enemy some where in the level, walking around a certain tile involve adding solid objects, etc. The n-gram model is used to specify these rules so instead of reacting to a single action, the system responds to a n-sequence of actions. During the generation the algorithm also keeps track of all the placed sprites to make sure it doesn't overpopulate the level. Then, the avatar sprite at the bottom center of the level is added.

Beaupre's constructive pattern generator: is a constructive algorithm that was designed for Beaupre et al. [1] work on analyzing the effect of using design patterns in automatic generation of levels for the GVGAI framework. In their work, they analyzed 97 different games from the GVGAI framework using a 3×3 sliding window over all the levels after transforming the levels to use sprite type from the *GameAnalyzer* instead of the actual sprites. They constructed a dictionary that contains all these different patterns (they discovered $12,941$ unique patterns) and classified it based on the type of different objects mixed in. There are border patterns which are patterns that appear on the border of the level, avatar patterns which are patterns that contain an avatar sprite in them, and others. The algorithm starts by checking if the game have solid sprites. If that was the case, it fills the border areas using border patterns. The rest of the level is picked randomly from the rest of the patterns based on their distribution while making sure there is only avatar in the level and the level is still fully connected. Finally, the system converts the generic

sprite types to specific game sprites while making sure that goal sprites numbers are larger than the specified number in the termination condition.

FrankfurtStudents constructive generator: is designed for GECCO 2018 level generation competition. This generator is more handcrafted generator that tries to identify the type of the game (racing game, catching game, etc) and based on that type defines the level size and gives each game sprite a preferred position with respect to other sprites and the level.

Search-based methods Search-based methods uses a search based algorithm such as genetic algorithm to find levels that are playable and more enjoyable than random placement of sprites. This section describes all the known search-based generators.

Amy12 genetic generator: is build on top of the sample genetic generator. This generator was submitted to IJCAI 2016 competition. The idea is to generate level that has a certain suspense curve. Suspense is calculated by measuring the number of actions that leads to death during the gameplay using OLETS agent. The algorithm tries to modify the level to get a suspense curve with three peak points of less than 50% height. The advantage of doing that that it makes sure that the generated levels are winnable as levels that are not winnable will have higher peaks in the suspense curve.

Jnicho genetic generator: uses a standard genetic algorithm compared to FI2Pop used in the sample generator [12]. The generator combines the constraints and relative algorithm performance in a single fitness function where the relative algorithm performance is measured between a MCTS agent and an OneStepLookAhead agent. The score values of these agents is normalized between 0 and 1 to make sure that the relative algorithm performance doesn't overshadow the rest of the constraints. This generator was submitted to IJCAI 2016 competition.

Number13 genetic generator: is a modified version of the sample genetic generator. This generator was submitted to IJCAI 2016 competition. The generator uses adaptive methods for crossover and mutation rates with a better performing agent than the modified OLETS. It also allows crossover between feasible and infeasible chromosomes which was not allowed in the sample generator.

Beaupre's evolutionary pattern generator: uses similar idea to the their constructive generator described in the constructive methods. This generator is a modified version of the sample genetic generator provided with the framework but with the chromosome represented as 2D array of patterns instead of tiles. In that case, they use their constructive approach to initialize the initial population.

Sharif's pattern generator: is submitted as a participant in GECCO 2018 competition. This generator similar to Beaupre's evolutionary pattern generator in

using identified patterns to build the level. In a previous work by Sharif et al. [17], they identified a group of 23 different unique patterns that they are using during the generation process. These patterns are selected to have similar meaning to the pattern identified in the work by Dahlskog and Togelius in generating levels for Super Mario Bros (Nintendo, 1985) [5].

Architect genetic generator: is the winner of GECCO 2018 level generation competition. The algorithm is build on the sample genetic generator provided with the framework, the only difference that it uses two-point crossover and a new constructive initialization technique. The new constructive technique is similar to the sample random generator with some improvements. It starts with calculating the size of the map, then building a level layout similar to the one used in the sample constructive technique, followed by adding an avatar to the map. Finally 10% of the map is picked randomly from all the possible sprites.

Luukgvgai genetic generator: is another GECCO 2018 submission, similar to the other is a modified version of the sample genetic generator using tournament selection and two-point crossover.

Tc_ru genetic generator: is a modified version of the sample genetic generator that uses eigth-point crossover and tournament selection instead of rank selection. It also used cascaded fitness where the relative algorithm performance has higher precedence than the unique interactions. This algorithm was submitted to GECCO 2018 GVGAI level generation competition.

Constraint-based methods Constraint-based methods used a constraint solver to generate levels. The idea of the generator is to find the right constraints needed to make sure the generated content is the targeted experience. So far only one generator has been developed for GVGAI using that technique.

ASP generator: [11] uses Answer Set Programming (ASP) to generate levels. The ASP programs are evolved using evolutionary strategy that uses relative algorithm performance between the *sampleMCTS* agent and the *sampleRandom* agent. The evolved rules are divided into three different types. The first type is a basic set of rules that make sure the generated level is not complicated such as making sure there is only one sprite per tile. The second type are more game specific rules based such as identifying the Singleton sprites in the current game. The third type is concerned with the maximum number of objects that can be produced for a certain sprite type.

1.3 Discussion

The presented generators differ in the amount of time needed to generate a level and the features of the generated content. The constructive generators take the least amount of time to generate a single level without a guarantee that the generated level is beatable. On the other hand, both search-based and constraint-based generators take longer time but generate challenging beatable levels as they use automated playing agents as a fitness function. The constraint-based generator only takes long time to find an ASP generator which could be used to generate many different levels as fast as the constructive generators, while search-based generators take a long time to find a group of similar looking levels.

Some of the presented algorithms were submitted to the GVGAI level generation track in IJCAI 2016 (Easablade, Amy12, Number13, and Jnicho genertors), CIG 2017 (N-Gram constructive generator), and GECCO 2018 (Architect, FrankfurtStudents, Sharif, Luukgvgai, and Tc_ru generators). These generators were compared with respect to each other during the competition.

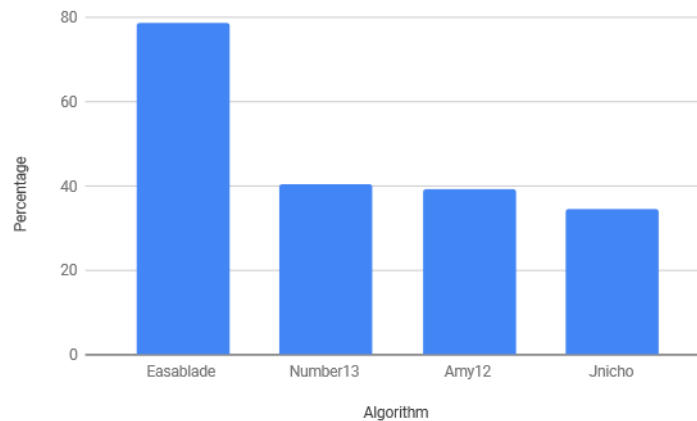Figure 3 shows the results from IJCAI 2016 competition.

Fig. 3: IJCAI 2016 Level generation track results.

The algorithms were tested against four GVGAI games:

– **Butterflies:** is a VGDL game about collecting all butterflies before they open all the cocoons in the level.
– **Freeway:** is a VGDL port of Freeway (David Crane, 1981). Similar to Frogs, the goal is to reach the other side of the road without being ran over by cars. The

difference is that this game is a score-based game where the player need to reach the goal more than one time to get a better score.

– **Run:** is a VGDL runner game where the player is trying to outrun a flood coming from behind and reach the exit door in time.

– **The Snowman:** in this game, the goal is to stack the snowman pieces (legs, trunk and head) in the correct order to create a snowman.

The results are very clear that Easablade beats the other three generators. A possible reason is that Easablade generated a few amount of sprites in the generated scene with a nice layout. On the other hand, all the generated levels were either unplayable (exits hidden behind a wall) or easy to beat (exits directly beside the player).
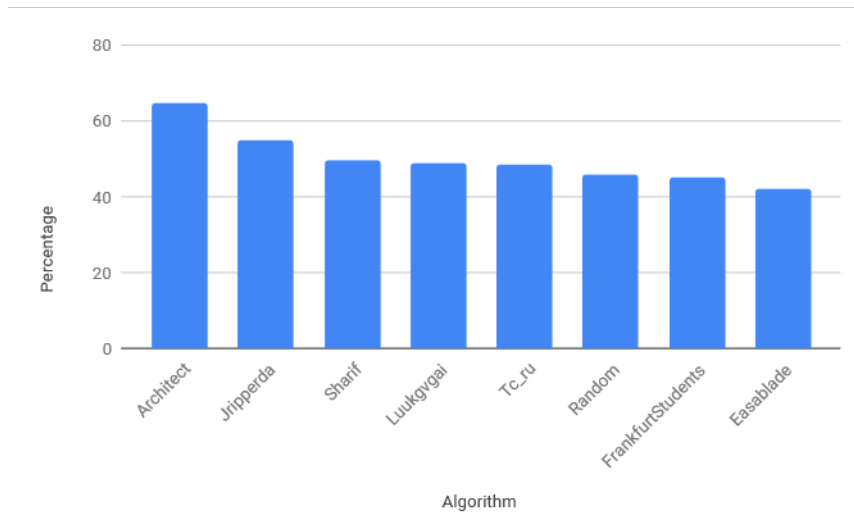

Fig. 4: GECCO 2018 level generation track results.

Figure 4 shows the results from GECCO 2018 competition. The algorithms were tested against three GVGAI games picked from stepheston et al. work [18] on finding the most discriminatory VGDL games.

– **Chopper:** is a VGDL action game where the player needs to collect bullets to kill the incoming tanks without getting killed by the bullets from the enemy tanks. Also, the player has to protect the satellites from being destroyed.

– **Labyrinthdual:** is a VGDL puzzle game where the player tries to reach the exit without getting killed by touching spikes by changing their color to pass through colored obstacles.

– **Watergame:** is a VGDL puzzle game about trying to reach the exit without drowning in water. To pass the level, the player has to push couple of potions around to convert the water to floor so the player can pass over it.

Architect won the competition with a small difference compared to Easablade. The reason behind the winning is not clear as some other generators have similar looking levels. Shockingly, Easablade got the worst rating compared to the random generator. We think that the choice of games affected the outcome of the competition. For example a puzzle game like Watergame is only playable if it is solvable having a big level and sparse object will be against the generator interest while in a game like freeway that might be better.

2   Rule Generation in GVGAI

The rule generation track was introduced in 2017 as a new competition track [7]. It is the second generation track for the GVGAI framework. The competition focuses on creating a new playable games for a provided game level with all the game sprites. In this track, every participant submits a rule generator that produces the interaction sets and termination conditions in a fixed amount of time. The framework provides the generator with a game level in form of 2D matrix of characters which can be translated to its corresponding game sprites using the provided level mapping. In return, the generator produces two arrays of strings that cover the game interaction set and termination conditions.

```
public abstract class AbstractRuleGenerator {

    public abstract String[][] generateRules(SLDescription sl, ElapsedCpuTimer time);

    public HashMap<String, ArrayList<String>> getSpriteSetStructure(){
        return null;
    }
}
```

Fig. 5: AbstractruleGenerator Class functions.

The game level and the game sprites are provided through the `SLDescription` object. Figure 5 shows the `AbstractRuleGenerator` class the user needs to extend to build their own rule generator. The user has to implement `generateRules` function and return the two arrays that contain the game interaction set and the termination

conditions provided the `SLDescription` object and `ElapsedCpuTimer` object. The SLDescription object provides the generator with a list of all game sprites. These game sprites have name, type, and related sprites. For example: In Zelda, shown in Listing 1, the player character is named `avatar` and it is of type `ShootAvatar` (which means it can shoot in all four directions) and has `sword` as related sprites (as it shoots it in any of the four directions). Beside the game sprites, the `SLDescription` object provides a 2D matrix that represents the game level where each cell represents the sprites that are present in that cell. When running the competition, the `ElapsedCpuTimer` provides the generator with five hours to finish generation.

The user can also override an optional function called `getSpriteSetStructure` which returns a hashmap between a string and an array of strings. The hashmap represents the sprite hierarchy required during the rule generation. For example: In zelda, the user can generate three rules that kill the *avatar* when it hits any `monsterQuick`, `monsterNormal`, and `monsterSlow` sprites or they can generate one rule that kills the `avatar` when it hits a `harmful` sprite and then define the `harmful` sprite in the sprite structure as all these three monsters.

Fig. 6: The provided level for Zelda.

The framework also provides the user with a `LevelAnalyzer` object that analyzes the provided level and allows the user to ask about sprites that cover certain percentages of the map and/or of a certain type. For example: the generator can get the `background` sprite by asking the `LevelAnalyzer` to get an `Immovable` sprite that covers 100% of the level. This information could be used to classify the game sprites to different classes. Table 3 shows some example classes that can be recognized from the game of Zelda presented in figure 6. These categories are defined based on gen-

eral game knowledge. For example: score objects such as coins in Mario covers small percentage of the level.

Class Type	Sprite Image	Sprite Type	Threshold	Surrounding Level
Background		Immovable	$\geq 100\%$	FALSE
Wall		Immovable	$< 50\%$	TRUE
Score/Spike		Immovable	$< 10\%$	FALSE

Table 3: The *LevelAnalyzer* data for the game of *Zelda*.

We ran the rule generation track twice: at CIG 2017 and GECCO 2018. In both times, we didn't get any submissions due to the lack of advertisement for the track and the harder the problem is to tackle (generating rules for a game is harder than just producing a level). In the following subsections, we will discuss the sample generator that were provided with the competition and the only other generator that we found in the literature which was created before the track existed.

2.1 Sample Generators

In similar manner to the level generation track, the rule generation track comes with three different generators: random, constructive, and search based generators. These generators increase in complexity and quality of the generated content.

Sample random generator The sample random generator is the simplest of all the generators. The generator just pick any random interactions and termination conditions that will compile into a VGDL game without any errors. The generator follows the following steps to generate a random VGDL game.

1. Pick a random number of interactions.
2. Generate a random amount of interactions by repeating the following steps until reaching the chosen amount of interactions while making sure it compiles with no errors:
   (a) pick two random sprites.
   (b) pick a random `scoreChange` value.
   (c) pick a random interaction.
3. Generate two terminal conditions for winning and losing.

- *Winning:* either a time out with a random time or a certain sprite count reaches zero.
- *Losing:* if the avatar dies, the game is lost.

Sample constructive generator The constructive generator is more sophisticated than the sample generator. It uses a template based generation to generate a good playable game. The generator has a game template that was designed based on game knowledge. For example: if there is a Non-Playable Character (NPC) running after a certain object, there is a high chance that it will kill it upon collision. The generator utilizes the sprite types from `SLDescription` object and sprite classes from `LevelAnalyzer` object to fill the template. The following are the steps taken by the sample generator.

1. **Get Resource Interactions:** collectResource interaction is added for all the resource sprites.
2. **Get Score and Spike Interactions:** each object has 50% chance to be either collectible or harmful. Collectible sprites are killed upon collision with the avatar and give one point score. Spike sprites kills the avatar upon collision.
3. **Get NPC Interactions:** different interactions are added for different types of NPCs. Usually they are either collected by the avatar for 1 score point or kill the player upon collision. Some NPCs could spawn other sprites which also could be either collectible or deadly.
4. **Get Spawner Interactions:** Similar to NPCs, spawner sprites decide if the generated sprites will kill the avatar or get collected for points.
5. **Get Portal Interactions:** if the portal is of type *door*, the avatar will destroy it upon collision. Otherwise, the avatar is moved to the destination portal.
6. **Get Movable Interactions:** the generator decides randomly if these movable objects are harmful or collectible sprites.
7. **Get Wall Interactions:** the generator decides if the wall sprites will be fire walls or normal walls. Fire walls kill any movable sprite upon collision, while normal walls prevent any movable sprite.
8. **Get Avatar Interactions:** This step only happens if there are any harmful sprites being generated and the avatar can shoot bullets. The generator adds interactions between the bullets and harmful sprites to kill both of them upon collision.
9. **Get Termination Conditions:**
   - *Winning:* The generator picks a random winning condition such as the avatar reached a door, all harmful sprites are dead, all the collectible sprites are collected, or time runs out.
   - *Losing:* the game is lost when the avatar dies.

Sample genetic generator Similar to the level generation track, the search based algorithm uses the FI2Pop algorithm [9] to generate interaction set and termination conditions. The infeasible chromosomes try to become feasible by satisfying these three constraints.

– Interaction set and termination conditions compile with no errors.
– A do-nothing agent stays alive for 40 frames.
– The percentage of bad frames are less than 30%, where bad frames are frames where one sprite or more are outside of the screen boundaries.

If the chromosome satisfies these constraints, it get moved to the feasible population. The feasible chromosomes try to improve their fitness. The feasible fitness consists also of three parts.

– Increase the relative algorithm performance [13] between three different agents (OLETS agent, MCTS agent, and random agent, in this order of performance).
– Increase the number of unique rules that are fired during the game playing session. Game rules are added to be used, if a rule is not being used by any agent then it violates this constraint.
– Increase the time the agent takes to win the level. We don't want games that could be won in less than 500 frames (20 seconds).

We used rank selection with 90% chance for crossover and 10% chance for mutation. We used one-point crossover to switch both interaction set and termination conditions. For the mutation, three different operators were used.

– **Insertion:** either inserting a new rule/termination condition or inserting a new parameter.
– **Deletion:** either delete a rule parameter or delete an entire rule/termination condition.
– **Modify:** either modify one of the rule parameters or modify the rule/termination condition itself.

We initialized the algorithm with 10 constructive chromosomes, 20 random chromosomes, and 20 mutated versions of the constructive chromosomes to a have a population of size of 50. We used 2% elitism which keeps the best chromosome between generations.

Pilot Study and Discussion We applied these three generators on three different games (Aliens, Boulderdash, and Solarfox) selected based on the different algorithm performance by Bontrager et al. work [2]. Our first study was to see the diversity
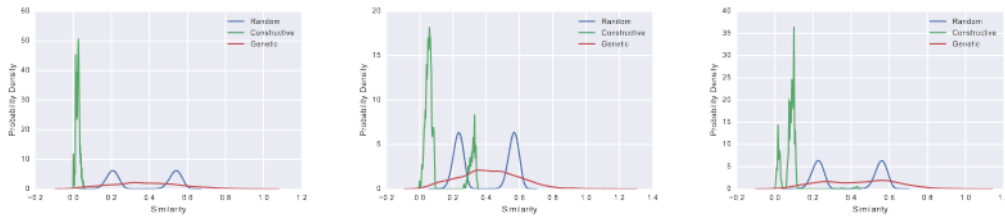
Fig. 7: Probability density of the similarity metric between the generated games. These graphs represent Aliens, Boulderdash, and Solarfox from left to right. "0" means the games are identical, while "1" means the games are totally different.

	Aliens	Boulderdash	Solarfox
Genetic vs Rnd	2/8	7/7	11/15
Genetic vs Const	0/14	8/14	6/18
Const vs Rnd	9/10	10/11	4/5

Table 4: Comparison between our rule generators where "Genetic" is the sample genetic generator, "Rnd" is the sample random generator, or "Const" is the sample constructive generator. The first value is the number of times the user preferred the first game over the second. The second value is the total number of comparisons.

of the generated content between these generator on these three games. Figure 7 shows the probability density function of how similar the generated games to each other where 0.0 means 100% similar and 1.0 means they are totally different. These distribution were calculated by generating 1000 games from both sample random generator and sample constructive agent and 350 generated games from the sample genetic generator (due to time constraints). From all distributions, we can see that the sample genetic generator is able to generate games that ranges from very similar to each other to totally different, while the constructive generator was very limited to similar games, which was expected as we are using a template with a small amount of parameters to be changed.

We also conduced a similar user study to the level generator track to compare these three generators to each other in terms of preference. The users were faced with two generated games with the same level layout from either the same generator or a different generator, and we asked them to pick which one they thought it was best (first or second), both of them being equally good, or neither of them being good. Table 4 shows the results of the user study. As we expected constructive generator and genetic generator are mostly preferred over the random generator except for Aliens in the genetic generator. The reason for aliens being different is the genetic

generator didn't have much time to evolve so to satisfy the constraints it generated a game with `undoAll` as an interaction between player and background to satisfy the bad frames constraint. `undoAll` interaction pauses the whole game, not allowing for anything to happen. Another remark is that the genetic generator was not preferred over the constructive one. We think the reason for that is the constructive templates were better designed and users never noticed they are similar to each other.

2.2 Other Generators

As far as we are aware, there is only one other work that was done towards rule generation, by Thorbjørn et al. [13]. The generator is similar to the sample genetic generator as the sample generator idea was based on that work. Both generators use the difference between the performance of different algorithms (relative algorithm performance) as an evaluation function. The difference is that this generator uses evolutionary strategies with mutation operators to generate an entire game instead of interaction set and termination conditions.

2.3 Discussion

The rule generation competition has been running for two years but no one has submitted yet to it. We think the reason behind that might be that the competition is harder than level generation (need more computations) and usually runs at the same time with level generation so the competitors either choose to participate in the level generation competition. We think that there is a huge opportunity of research in rule generation competition. Some researchers work on generating full games [19] based on Ralph Koster Theory of Fun [10], generating games by modifying the game code [4], generating games based on a single work [3], etc.

References

1. S. Beaupre, T. Wiles, S. Briggs, and G. Smith, "A Design Pattern Approach for Multi-Game Level Generation," in *Artificial Intelligence and Interactive Digital Entertainment*. AAAI, 2018.
2. P. Bontrager, A. Khalifa, A. Mendes, and J. Togelius, "Matching Games and Algorithms for General Video Game Playing," in *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2016.
3. M. Cook and S. Colton, "A rogue dream: Automatically generating meaningful content for games," in *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2014.
4. M. Cook, S. Colton, A. Raad, and J. Gow, "Mechanic miner: Reflection-driven game mechanic discovery and level design," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2013, pp. 284–293.
5. 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*. ACM, 2012, p. 1.

6. L. Johnson, G. N. Yannakakis, and J. Togelius, "Cellular automata for real-time generation of infinite cave levels," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM, 2010, p. 10.

7. A. Khalifa, M. C. Green, D. Pérez-Liébana, and J. Togelius, "General Video Game Rule Generation," in *IEEE Conference on Computational Intelligence and Games (CIG)*, 2017.

8. 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*. ACM, 2016, pp. 253–259.

9. S. O. Kimbrough, G. J. Koehler, M. Lu, and D. H. 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, pp. 310–327, 2008.

10. R. Koster, *Theory of fun for game design*. " O'Reilly Media, Inc.", 2013.

11. X. Neufeld, S. Mostaghim, and D. Perez-Liebana, "Procedural level generation with answer set programming for general video game playing," in *Computer Science and Electronic Engineering Conference (CEEC), 2015 7th*. IEEE, 2015, pp. 207–212.

12. J. Nichols, "The Use of Genetic Algorithms in Automatic Level Generation," Master's thesis, University of Essex, 2016.

13. T. S. Nielsen, G. A. Barros, J. Togelius, and M. J. Nelson, "General video game evaluation using relative algorithm performance profiles," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2015, pp. 369–380.

14. D. Perez-Liebana, S. Samothrakis, J. Togelius, T. Schaul, S. M. Lucas, A. Couëtoux, J. Lee, C.-U. Lim, and T. Thompson, "The 2014 general video game playing competition," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 8, no. 3, pp. 229–243, 2016.

15. N. Shaker, A. Liapis, J. Togelius, R. Lopes, and R. Bidarra, "Constructive generation methods for dungeons and levels," in *Procedural Content Generation in Games*. Springer, 2016, pp. 31–55.

16. N. Shaker, J. Togelius, and M. J. Nelson, *Procedural content generation in games*. Springer, 2016.

17. M. Sharif, A. Zafar, and U. Muhammad, "Design Patterns and General Video Game Level Generation," *Intl. Journal of Advanced Computer Science and Applications*, vol. 8, no. 9, pp. 393–398, 2017.

18. M. Stephenson, D. Anderson, A. Khalifa, J. Levine, J. Renz, J. Togelius, and C. Salge, "A continuous information gain measure to find the most discriminatory problems for ai benchmarking," *arXiv preprint arXiv:1809.02904*, 2018.

19. J. Togelius and J. Schmidhuber, "An Experiment in Automatic Game Design," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG)*, 2008, pp. 111–118.

20. G. Turk, "Generating textures on arbitrary surfaces using reaction-diffusion," in *ACM SIGGRAPH Computer Graphics*, vol. 25, no. 4. ACM, 1991, pp. 289–298.