Procedural Level Generation with Answer Set Programming for General Video Game Playing

Xenija Neufeld
Otto von Guericke University Magdeburg
Institute of Knowledge and Language Engineering
Magdeburg, Germany
xenija.neufeld@st.ovgu.de

Sanaz Mostaghim
Otto von Guericke University Magdeburg
Institute of Knowledge and Language Engineering
Magdeburg, Germany
sanaz.mostaghim@ovgu.de

Diego Perez-Liebana
University of Essex
School of Computer Science and Electronic Engineering,
Colchester, United Kingdom
dperez@essex.ac.uk

***Abstract*—This paper proposes an automatic way of evolving level generators for arbitrary 2D games, which are described in the Video Game Description Language (VGDL). The process works as follows: a game described in VGDL is interpreted and transformed in a set of rules defined in Answer Set Programming (ASP), along with other general and customizable rules. Although a set of rules described in ASP can generate multiple levels, not all of them will be playable or well designed. Therefore, an evolutionary process is run to determine the values of the parameters of those customizable rules. The different level generators are evaluated with general video game playing agents, which are able to play any game and level in the framework. The aim is to maximize the difference between their performance in the levels generated, under the assumption that levels are better designed if good skilled players play better than poor agents. This work presents some initial experiments that suggest that it is possible to evolve interesting level generators using this technique, and outlines some lines of future work.**

I. Introduction

There are many different approaches that can be used to generate levels for video games. Most of them are search-based and therefore need a fitness-function for evaluating the created levels. These approaches show good results in generating levels for the game they were created for. However, they cannot be used for games with different game mechanics, rules or aims, since the search space, the fitness function and the content representation could change in these cases.

To overcome these difficulties while building a general level generator, a method is needed that can specify the desired properties of the content and the search space without being bound to specific game rules. In this work, we propose a fully automatic general level generator that is designed to read descriptions of arbitrary games written in Video Game Description Language (VGDL). We use a combination of Answer Set Programming (ASP) and an Evolutionary Algorithm (EA). The former is used to generate maps for different games of the GVG-AI framework, whereas the latter optimizes the difficulties of the levels.

Recently, it has been shown that ASP can be used to create not only new game mechanics [1] but also maps for mazes and dungeons [2], [3] and puzzle games [4]. Its advantage is that the game mechanics and the structure of the content can be defined through logical expressions and the search space can be easily reduced by constraints, so that no fitness function is required at this point. Furthermore, style constraints can be added to increase the aesthetic value of the maps generated.

In this paper, we develop a generator using VGDL within the GVG-AI framework. The framework already offers descriptions of 40 different Atari-like games, and further games can be added following the rules of VGDL. In the work proposed here, these game descriptions are converted into ASP rules. It is important to highlight the limitations and difficulties of this conversion, since the generator is kept general and cannot identify all relationships between game objects as it is done in [2]–[4].

While creating ASP rules, we integrate some of the *Visual Impression Distance* measures (such as vertical and horizontal balance) introduced in [5]. We assume that optimizing these measures should improve the visual impression of the levels. Then, we add some additional random constraints for the unknown characteristics of game objects to the ASP program. That way, we guide the generator towards different solutions in the search space.

Afterwards, we test the generated maps by letting several agents with different skill levels play them. This way, we can test whether the levels are solvable. Furthermore, we use the difference between the agents performance to distinguish between levels, assuming that this difference should be higher for better levels. This assumption is based on the findings of [6], where rules for multiple games were evolved, and the difference of performances between agents was higher for well-designed than for worse-designed games. In general, one may assume that levels where this difference is high require more amount of skill to be solved than those where both good and a poor agents behave similarly [7]. Finally, we evolve the levels by mutating the randomly added constraints, searching for maps with higher performance differences.

This paper is structured as follows. Section II introduces some background on VGDL and ASP. Then, Section III details the process of generating levels with ASP, followed by the evolution of level generators in Section IV. Section V shows some results, and finally Section VI concludes the paper and outlines potential future work.

```
BasicGame
    SpriteSet
        base    > Immovable    color=WHITE img=base
        avatar  > FlakAvatar   stype=sam
        missile > Missile
            sam > orientation=UP color=BLUE singleton=True img=spaceship
            bomb > orientation=DOWN color=RED  speed=0.5 img=bomb
        alien   > Bomber stype=bomb   prob=0.01  cooldown=3 speed=0.8 img=alien
        portal  >
            portalSlow > SpawnPoint stype=alien cooldown=16 total=20 img=portal
            portalFast > SpawnPoint stype=alien cooldown=12 total=20 img=portal

    LevelMapping
        0 > base
        1 > portalSlow
        2 > portalFast

    TerminationSet
        SpriteCounter      stype=avatar       limit=0 win=False
        MultiSpriteCounter stype1=portal stype2=alien limit=0 win=True

    InteractionSet
        avatar  EOS  > stepBack
        alien   EOS  > turnAround
        missile EOS  > killSprite
        missile base > killSprite
        base bomb > killSprite
        base sam > killSprite scoreChange=1
        base   alien > killSprite
        avatar alien > killSprite scoreChange=-1
        avatar bomb  > killSprite scoreChange=-1
        alien  sam   > killSprite scoreChange=2
```

Fig. 1: Example of VGDL description of the game Aliens.

II. Background

In this section, we describe VGDL, provide a short description of the GVG-AI Framework and briefly explain the concepts behind ASP.

A. Video Game Description Language (VGDL)

VGDL is a language that allows the description of many 2D games, and it was originally implemented by Tom Schaul in Python [8]. Every game object can be defined as a sprite in the *SpriteSet*, including some properties like orientation and movement abilities. Furthermore, a *LevelMapping* section contains the representation of each object on the 2D map. The *InteractionSet* defines which effects are applied to objects when they collide, and finally the *TerminationSet* defines which conditions have to be met for the game to end. The description of the game *Aliens* (based on *Space Invaders*) is shown in Figure 1 as an example.

B. GVG-AI Framework

The GVG-AI framework was developed for the General Video Game AI Competition [9] as an environment for agents that should be able to play any game that is given to them. Currently, the framework provides 40 manually designed games that are described in VGDL. The framework, games and agents are implemented in Java.

C. ASP

Answer Set Programming (ASP) is a declarative programming method that can be used to describe *what* problem is to be solved instead of *how* it is to be solved. The problem can be expressed through logical terms, which are then passed to a solver that provides answer sets using deductive reasoning. The logical expressions used hereby can be atoms (simple statements), predicates (statements with parameters), choice rules (allowing a choice of elements) or integrity constraints (allowing conditions).

```
1.  dimX(1..10).
2.  dimY(1..10).
3.  tile((X,Y)) :- dimX(X), dimY(Y).

4.  maxdimX(X) :- dimX(X), not dimX(X+1).
5.  maxdimY(Y) :- dimY(Y), not dimY(Y+1).
6.  mindimX(X) :- dimX(X), not dimX(X-1).
7.  mindimY(Y) :- dimY(Y), not dimY(Y-1).

8.  sprite((X,Y), wall) :- tile((X,Y)), mindimX(X).
9.  sprite((X,Y), wall) :- tile((X,Y)), maxdimX(X).
10. sprite((X,Y), wall) :- tile((X,Y)), mindimY(Y), not mindimX(X), not maxdimX(X).
11. sprite((X,Y), wall) :- tile((X,Y)), maxdimY(Y), not mindimX(X), not maxdimX(X).

12. 0 {sprite(T, wall; portalFast; portalSlow; base; avatar)} 1 :- tile(T).

13. :- sprite((X,Y), portalFast),  not mindimY(Y-1), tile((X,Y)).

14. :- sprite((X,Y), avatar),  not maxdimY(Y+1) , tile((X,Y)).

15. :- not 1 {sprite(T, avatar) : tile(T)} 1 .

16. :- not 1 {sprite(T, portalFast) : tile(T)} 1 .
```

Fig. 2: Example of ASP description of the game Aliens.

Answer sets delivered by a solver are guaranteed to satisfy all constraints defined in the problem description. That way, the design space can be incrementally specified from which undesired answers are excluded. For this reason, ASP can be used for procedural content generation without using any evaluation function. Nevertheless, the desired answers can only be produced if the game logic and all dependencies between game objects are completely known and are correctly expressed in ASP. Some detailed explanations on building the design space for a game are described in [10].

For example, if a level for the game *Aliens* should be produced, the rules shown in Figure 2 could be sufficient for the ASP solver to deliver acceptable maps. Here, lines 1–3 define the dimensions of the map saying that each grid cell is a tile. Lines 4–11 define the borders of the map which is surrounded by *walls*. Line 12 indicates that each tile has at most one of the given sprites. Lines 13 and 14 define the y-positions of the *portal* (which spawns the aliens) and the *avatar*. Here, the portal is on the top of the level and the avatar on its bottom, as this is the common structure of a level of *Aliens*. The last two lines specify that there is exactly one avatar and exactly one portal in the level. Since there is no constraint on the number or position of the *base* sprites (wall blocks), they can be created on any tile that has no other sprite on it.

The ASP program shown was created ad hoc, with the rules of the game known by the authors. However, this example points out the problem of automatically mapping VGDL rules into an ASP program. Especially the last four lines could not be read out from the VGDL-description shown in Figure 1. There is no information given about the amount or the position of any object and thus, it is not possible to know that the portal should be placed on the opposite side of the avatar, or that there should be only one of each.

III. Map generation through ASP

Having introduced the basic elements of our level generator, we now describe how the levels are created. Although not all the information about game rules is available, it is still possible to infer some dependencies between game objects from the VGDL representation of the games. For that reason,

the main step towards generating levels for a given game is the conversion of the traceable information into ASP rules as far as it is possible. For a better handling of the game data we use the GVG-AI framework that gives an access to each sprite, interaction and termination through their Java class representations.

A. Basic ASP rules

Before concentrating on game specific ASP constraints, we create some basic rules that are common for every game and can be used in further statements. These include definitions like the (Manhattan) distance measure and adjacency of tiles. Furthermore, the reachability of a tile *A* from a tile *B* is defined here, saying that a tile is reachable from another one if there is a path between them consisting of only passable tiles. Tiles that are not passable are defined later within game specific rules. An additional constraint is generated here, allowing to have at maximum one sprite on a tile.

Moreover, the borders of a map are defined here using the dimensions given by the user. Since without loss of generality levels of all considered games are surrounded by walls, we also create the corresponding rules at this point.

B. Game specific ASP rules

The most challenging part of this work is the creation of game specific rules. Here, one important task of the generator is the identification of the desired amount of each game object. Sometimes, this information can be read out directly from the characteristics of sprites given in VGDL. For example the game object *sam* is marked as a singleton in Figure 1, so its amount can directly be set to one. Nevertheless, in most cases, no information about the number of sprites of each type is given. Therefore, the generator needs to check the termination conditions for them. The following example illustrates the difficulty of this process.

In many cases, a termination condition for a game is a *(Multi)SpriteCounter*. This clause specifies the number of sprites of certain game objects that should be reached for the game to end. For example the line "MultiSpriteCounter stype1=portal stype2=alien limit=0 win=True" in Figure 1 says that the game can be won when there is no alien and no portal anymore. This indicates that at the beginning of the game, there should be *at least* one sprite of any of these two types.

However, it is still not possible to use this information directly, because neither *portal*, nor *alien* are listed in the *LevelMapping*, so no ASP constraints should be created for them. Thus, the generator has to find their relationships with other game objects to recognize that an alien is spawned by a portal and a portal has two different subtypes (*portalSlow* and *portalFast*). This means that there should be *at least* one of the two sprite types, portalFast *and* portalSlow. To make sure that the right ASP constraints are created for all game objects, the minimum and maximum limits are saved for each object, provided they are found in the description.

Another type of a relation can be a *key-lock*-dependency. Many games contain a mechanic where locks/doors have to be opened with keys that the player has to pick up before. Having recognized a key-lock pair in the VGDL description, the generator should create *the same amount* of both. Such type of dependency is also saved for each of the involved object types.

Furthermore, if there is a door and a key that opens it, it is very likely that there is some third object hidden behind the door (e.g. an exit). In most cases, the third object can be found in the *TerminationSet* (e.g. there would be a SpriteCounter for the exit). Thus, the generator should ensure that whatever is behind the door should not be reachable by the avatar at the beginning of the game. For that reason, one important ASP rule is the definition of *impassable* tiles.

In most games provided by the GVG-AI framework, walls are not passable by the avatar. However, this is not true for some games where the avatar can destroy them. Therefore, the generator checks the *InteractionSet* for those sprites that the avatar can collide without destroying them, and that would make the avatar to step back. Moreover, sprites that are not moveable but can kill the avatar are also regarded as impassable. One example of such a sprite could be a fire or a water tile. These sprites are then added to the definition of an *impassable* tile and can be used to make sure that certain objects are not reachable by the avatar. If there is a door in the game, it should be placed in such a way that after it is opened, the object behind it becomes reachable. Having identified these relationships, the generator adds appropriate rules to the ASP program.

In addition, some rules are added here that optimize the *Visual Impression Distance* measure of each object. Currently, there are two rules implemented providing that *vertical* and *horizontal balance* are sustained in every level for every object type. For the former, a rule is created that minimizes the difference of the number of certain game objects on the left half and those on the right half of the map. The latter rule is created similarly for the upper half and lower half. It is conceivable that further measures introduced in [5] can be added here for optimizing the maps.

ASP allows the optimization of multiple functions at the same time. Hereby, the importance of each function can either be set by its priority, or by setting weights for each function and optimizing the weighted sum. Since the balance of all objects is regarded equally important, all balance functions get the same priority. These optimization functions are treated as soft constraints in ASP. Therefore, they are not conflicting with hard constraints and can be used even if there is a rule that e.g. says that the enemies should spawn only in the upper half of the map.

C. Additional ASP rules

Although some important ASP rules are created in the previous steps, many properties of game objects are still not traceable by the generator due to its generality. For example, Figure 3 left shows that multiple portals and too many bases are created for the game Aliens. This comes from the fact that there is no upper limit given on the number of portals or base

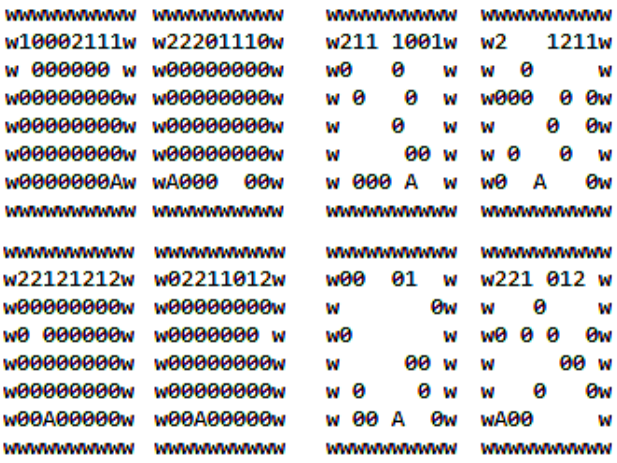


Fig. 3: Example maps for Aliens. Left: results of basic and game specific rules; right: results after adding horizontal/vertical balance rules and number constraints. (0 - base; 1 - portalSlow; 2 - portalFast; A - avatar; w - wall)

sprites in a level. A human designer would recognize that a high number of portals is too difficult for the player to handle. Only one or two portals would be enough, since each of them spawns 20 aliens.

For such cases, we propose limiting the number of game objects that have no limits given through game dependencies yet. Therefore, the generator is supposed to examine each object as well for lower limits as for upper limits and add missing rules. As a lower limit, we propose creating *at least* one object of each type, and as an upper limit *at most* one fourth of the map size. In case there are several subtypes of an object, they should be handled in one rule. So, e.g. it is known that there should be *at least* one of the two portalFast *or* portalSlow in *Aliens*. Although it is not known how many of them should be present *at most*. Having a map size of 80 grid cells, the generator creates a rule defining an upper limit of 20 for *both* of them *together*.

Limit constraints are one possibility to decrease the search space and exclude levels that are very likely to be unplayable. At the moment, these are the only additional rules implemented in the generator. Although other constraints could be added maintaining dependencies between arbitrary game objects e.g. ensuring that object *A* is reachable by object *B*.

IV. Evolution of ASP constraints

A. Evaluation

Even with all constraints described in the previous section, not all generated maps are solvable or have the desired difficulty. To find rulesets that generate solvable levels that are interesting to play, we test them letting agents play each map multiple times. Therefore, we use the *sampleMCTS* controller as one that is supposed to have a higher skill level in playing games. The *random* controller is used as a worse-playing agent.

For each ASP ruleset, we measure the difference between the average scores achieved by each agent while playing the maps that were generated from this ruleset. We add a score bonus of 1000 points every time an agent wins the game to favour solvable levels. Maps with higher score differences are assumed to have a better difficulty meaning that the more intelligent agent could handle the game better than the one applying random actions. Similarly, levels with lower differences are assumed to have a worse difficulty level, meaning that either the level was too hard or too easy for both agents. At this point, the validation depends strongly on the quality of agents, implying that meaningful results can only be achieved for those games that are playable by the agents.

We generate small populations of 10 rulesets/individuals with 5 levels each, since playing each map e.g. 3 times by two agents may take up to hours. At this point, the evaluation of a map with the help of a human could be a faster method. Nevertheless, we aim to have a completely automatic generator and use the agents limiting the maximum number of game steps to 1000 (at the GVG-AI Competition, each game lasts for max. 2000 steps). Additionally, an archive is kept with the best 10 individuals found during all generations. Note that this has a double purpose: apart from allowing us to determine the evolution of the rulesets, it also permits saving the best (but different) level generators found during the evolutionary process.

B. Mutation of additional ASP rules

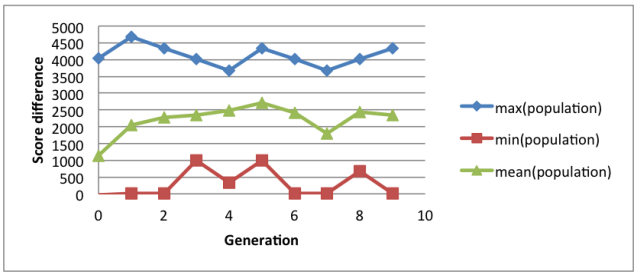
As already described in section III-C, the generator adds missing rules defining amount limits for all game objects. Nevertheless, it can be seen on the right part of Figure 3 that with an upper limit of one fourth of the map size, there are still too many portals (represented by 1 and 2) in the level of *Aliens*. So, it is desired to decrease their upper limit.

With the aim to change the shape of the search space and to guide the solver into different directions, we propose evolving the rule set using a mutation operator. For that purpose, each additional rule described in section III-C has a 50% chance of being mutated. Thus, one of the limits is changed to a random value between 1 and one fourth of the map size, always ensuring that the upper limit is equal or higher than the lower limit. The difficulties of resulting maps are evaluated as described in previous section. A (μ+λ) strategy is followed, and elitism is used to keep the best solution of the population. We also save the best solutions found so far on each generation to build an archive of generators.

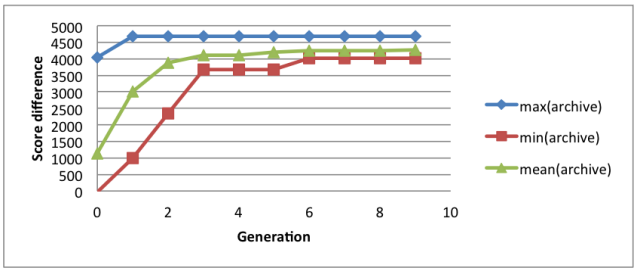
V. Results

Some interim results are shown in Figure 4. Here, we used three games in total, and we show one per row. Each of them was evolved over 10 generations $\times$ 10 rulesets (with each 5 levels) $\times$ 3 repetitions. The described mutator was applied using a (μ+λ)-selection with $\mu = \lambda = 10$.

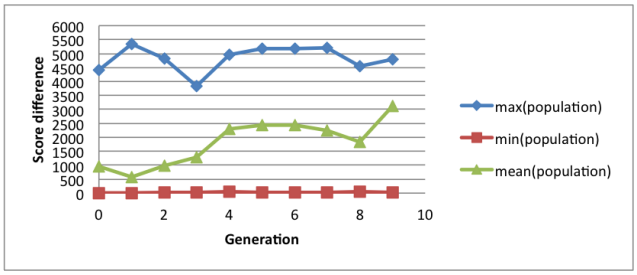
Although a fast increase of score differences between the two game-playing agents can already be seen in Figure 4, there is still room for improvement. Especially, the conversion of



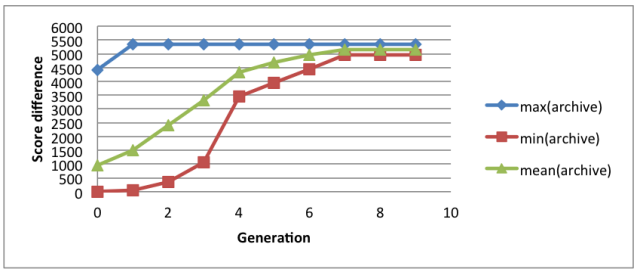
(a) Chase - Population Average



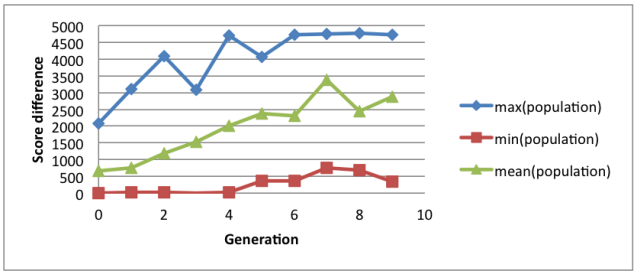
(b) Chase - Archive



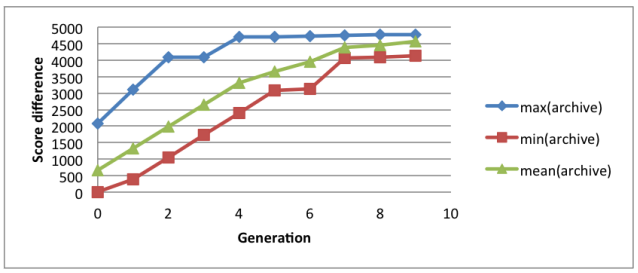
(c) Survive Zombies - Population Average



(d) Survive Zombies - Archive



(e) Zelda - Population Average



(f) Zelda - Archive

Fig. 4: Score differences for the games *Chase*, *Survive Zombies*, *Zelda*, one per row. Fitness is the average difference of scores between the two agents. On the left column, the minimum, average and maximum fitness in the population per generation is shown. The right column shows the min, average and maximum fitness of the individuals present in the archive at each generation, as described in section IV-A.

VGDL descriptions into ASP rules needs to be complemented. For example, Table I shows that the number of *keys* in the game *Zelda* can reach up to 7 and the number of *goals* can reach up to 11. This is not very conventional, since having collected only one key, the player is able to open any door (according to the given game description). In that case, it would have been enough to have only one of each object in the level.

However, the results show that in most cases, the amounts of game objects are acceptable. The number of e.g. *scared* entities in *Chase* can vary between 1 and 9 which can be handled by the player on a map size of 10×8 cells. Also the wall blocks whose amount can reach up to 9 (32 blocks are used to define the map borders) provide enough obstacles for the *scared* entities to run and hide from the player.

Furthermore, all generated levels show an equal distribution of game objects having good vertical and horizontal balance. An example of a level generated for *Survive Zombies* is given in Figure 5.

VI. Conclusions and Future Work

This paper presents an automatic way of evolving rule sets for level generators, for any game described in the Video Game Description Language (VGDL). The definitions of sprites and interactions that govern the game are parsed into Answer Set Programming (ASP), which is a declarative programming method that provides answer sets by means of deductive reasoning. The rules provided by ASP are then evolved in order to obtain interesting level generators, using the performance

Game	Game Object	Min. number	Max. number
Chase	scared	1	9
	wall	32	41
Survive Zombies	zombies	1	4
	slowHell	1	4
	fastHell	1	1
	honey	7	7
	flower	1	12
	wall	32	37
Zelda	monsterQuick	1	1
	monsterSlow	3	5
	monsterNormal	1	3
	key	1	7
	goal	1	11
	wall	32	34

TABLE I: Amount values for game objects of the three games evolved by the generator

```
wwwwwwwwww
ww0 1 -21w
w0-0  .-0w
w. .   00w
w-0 00   w
ww0..A00.w
w11   .  w
wwwwwwwwww
```

Fig. 5: An example of Survive Zombies level generated by the proposed approach. (0 - flower; 1 - slowHell; 2 - fastHell; . - honey; - - zombie; A - avatar; w - wall)

of different general video game playing agents as a way to measure this. Concretely, we assume that a feasible and well designed level would differentiate between poor and good agents, an hypothesis that can be found in the literature [6], [7].

The work proposed here shows how it is possible to parse and evolve rules for a level generator for three different games in the GVGAI framework. The generated levels have a structure similar to many of the existing levels, although in some cases the number of game objects could be more fine-tuned.

Furthermore, albeit the Evolutionary Algorithm (EA) employed in this study is a simple approach, it still shows good performance in the games tested, even considering that not many generations were run due to long execution times. Our initial piece of future work consists of extending this study with longer execution times, bigger population sizes and also to all games in the GVGAI framework. Another piece of future work would involve trying new evolutionary approaches and the addition of new mutation operators, like the ruleset mutator introduced in [11], which would effectively increment and decrement the number of additional rules in the ASP program.

Moreover, one possibility consists of co-evolution between level generators and general video-game bots aiming at an improvement of performance for those games that are not yet playable by the current agents. Additionally, there is scope for other techniques such as Multi-Objective EA that optimizes several metrics of the levels (e.g. agent performance, game-play duration, player experience, a measure of beauty, etc.). Finally, there are also plans in place to perform human evaluations of the generated levels, in order to search for further quality measures.

REFERENCES

[1] A. Zook and M. O. Riedl, "Automatic game design via mechanic generation," in *Proceedings of the 28th AAAI Conference on Artificial Intelligence*, 2014.

[2] M. J. Nelson and A. M. Smith, "Asp with applications to mazes and levels (draft)," in *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, N. Shaker, J. Togelius, and M. J. Nelson, Eds. Springer, 2015.

[3] A. M. Smith and M. Mateas, "Answer set programming for procedural content generation: A design space approach," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 3, no. 3, pp. 187–200, 2011.

[4] A. M. Smith, E. Butler, and Z. Popovic, "Quantifying over play: Constraining undesirable solutions in puzzle design." in *FDG*, 2013, pp. 221–228.

[5] M. Preuss, A. Liapis, and J. Togelius, "Searching for good and diverse game levels," in *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*. IEEE, 2014, pp. 1–8.

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

[7] D. Perez, J. Togelius, S. Samothrakis, P. Rohlfshagen, and S. Lucas, "Automated Map Generation for the Physical Travelling Salesman Problem," *IEEE Transactions on Evolutionary Computation*, vol. 18:5, pp. 708–720, 2014.

[8] T. Schaul, "A video game description language for model-based or interactive learning," in *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*. IEEE, 2013, pp. 1–8.

[9] D. Perez, S. Samothrakis, J. Togelius, T. Schaul, S. Lucas, A. Couetoux, J. Lee, C. Lim, and T. Thompson, "The 2014 general video game playing competition," *Computational Intelligence and AI in Games, IEEE Transactions on*, 2015. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7038214

[10] A. M. Smith and M. Mateas, "Variations forever: Flexibly generating rulesets from a sculptable design space of mini-games," in *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*. IEEE, 2010, pp. 273–280.

[11] C.-U. Lim and D. F. Harrell, "An approach to general videogame evaluation and automatic generation using a description language," in *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*. IEEE, 2014, pp. 1–8.