Game Generation and Learning using VGDL and GVGAI

Mateusz Blommaert & Mats Ruell

Abstract

This paper discusses our creative part for the course Capita Selecta Artificial Intelligence: Contemporary AI. We took a look at the GVGAI framework and created our own versions of two existing algorithms to generate our own levels and rules. These levels are compared to example levels by letting several deep reinforcement learning algorithms train on them. Our results show that it is quite a hard task to get playable games and to properly train agents. We conclude that we did a decent job with our experiment, but that further improvement, work and research is very much possible.

1 Introduction

The General Video Game AI (GVGAI) competition has been ran since 2014. This competition offers a way for researchers and practitioners to compare their General Video Game Playing (GVGP) algorithms. The competition was run by five authors: Diego Perez, Spyridon Samothrakis, Julian Togelius, Tom Schaul and Simon M. Lucas; who also designed all the games and implemented the GVGAI framework [Diego Perez, 2015]. These games are all written in the Video Game Description Language (VGDL) [Ebner *et al.*, 2013]. This language describes games in a concise manner, where each game only takes a few lines of plaintext which is easily understandable for any human. The framework separates the problem of generating games into smaller sub problems such as generating the rules of the game [Ahmed Khalifa, 2017] and generating the level itself [Ahmed Khalifa and Togelius, 2016]

The official website for GVGAI[1] provides a 'getting started' guide on how to use their framework. They also provide several examples to help you understand how to generate your own games and create your own agents. This is where our creative part kicks off.

This papers starts by giving a short description of our creative part. In the Method section we go more into detail about every aspect of our methods. In Experiment we discuss the setup of our experiment and we end the paper by discussing the results.

[1] www.gvgai.net

2 Creative Part

Our creative part consists of multiple sub parts. At first, we adjust an existing Level Generator to generate new levels for existing games. We also adjust an existing Rule Generator to generate new rules for a game and thus effectively create new games. We started from the Constructive and Genetic Generator, which we both improved to generate better content. Afterwards we ran an experiment where we trained several stable baselines reinforcement learning agents with these generations. The end result compares these trained agents in a test set of generations.

3 Method

In this section we describe the methods applied in the context of generating levels, generating rules and training an agent.

3.1 Level Generator

The goal of the level generation track is to generate playable levels when provided with a description of a game in VGDL. It is important to mention that this description can be of any game. The level generation track, as well as the rule generation track, do not focus on one specific game and can be applied to a set of games. The framework contains four standard level generators, these are the random, constructive, search-based and the constrain-based generator [Drageset *et al.*, 2019]. New methods use a variation of one of these models. In our work we modified the constructive and genetic generator (a search-based method).

Constructive Generator

Constructive methods use some game knowledge to generate levels in order to, for example, not place the enemies to close to the player avatar. The standard constructive generator works in a six-step fashion. It starts with a pre-processing step where the percentage of tiles to be covered with sprites is calculated. Second, it builds the level layout. The next three steps add different objects to the game, these are respectively the avatar, the harmful sprites, collectibles and other sprites. Finally, it adds the goal sprites, which are necessary to reach the terminal condition of the game. Our new and improved

version of this algorithm contains four steps. At first, it calculates the map size. It then builds a level layout like in the standard constructive generator. The third step is the same as before, we add the avatar. The last step, however, is different as the algorithm fills ten percent of the level with random sprites. Figure 1 shows the process of the constructive generator algorithm applied on the Zelda game.

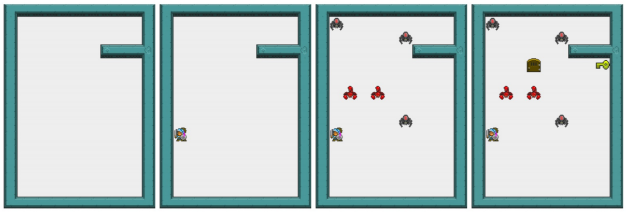

Figure 1: Process of the constructive generator algorithm applied to the game of Zelda

Genetic Generator

The standard genetic generator is based on the Feasible Infeasible 2-Population genetic algorithm (FI2Pop) [Kimbrough *et al.*, 2008]. This algorithms makes use of two populations, the infeasible and feasible population. The first contains the population not satisfying the problem constraints, while the latter is used to improve the fitness. The fitness function can be seen as an objective function in order to represent how 'fit' or how good the solution is. To calculate the fitness function two parameters are determined. First, we calculate the amount of unique interaction, as interesting playable games should have a good amount of unique interactions. Second, it takes the relative performance of the planner/learner into account; a game has a high performance when it there is a big difference between a good planner/learner playing the game versus a bad one. When during level generation, a chromosome does not satisfy the constraints, it is transferred to the infeasible population and vice versa.

We made three different changes to the algorithm. First, we used 2-point crossover instead of 1-point crossover. The crossover operator is used to create new solutions from the existing population, i.e. it combines the genetic information from two parents. In single-point crossover we choose one point in the parents' chromosomes. By exchanging the genes after the crossover point, the new solution (children) will contain information from both parents. When we apply two-point crossover, two random crossover points instead of one are chosen in the parents' chromosomes. Figure 2 depicts the difference between both crossover methods. The main reason to choose a two-point crossover over a one-point crossover, is to create more diverse offspring; in single-point crossover the offspring are fairly similar to their parents. Since we want to generate more diverse levels instead of similar boring levels, it is more desirable to use a k-crossover operator, such as 2-point crossover.

Second, we applied tournament selection instead of rank selection. In rank selection, every chromosome first gets a

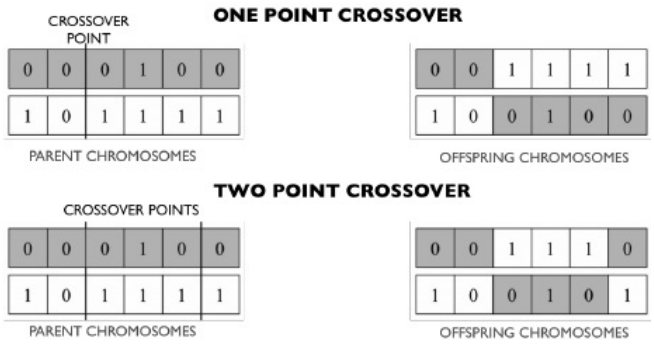

Figure 2: Single-point versus two-point crossover

rank; a new fitness is then assigned to each chromosome based on their ranking. In tournament selection we select k individuals from the population and let them 'compete' with each other. Figure 3 depicts this process. Several variations on tournament selection exist. Commonly, the best individual is chosen from this tournament, however, we decided to opt for a variation on this where we select the best individual with a probability of ninety percent. This way, the weaker individual has a ten percent chance to become the next parent. By allowing the weaker chromosome to win some tournaments we allow for a larger diversification of the population, and, moreover, we avoid getting trapped in local optima.

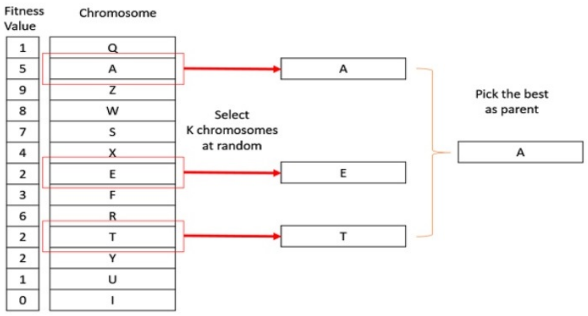

Figure 3: The tournament selection operator

Third, we applied our improved constructive generator for initializing the population used by the genetic generator.

3.2 Rule Generator

In the rule generation track the objective is to generate the rules to get playable games. Similar to the level generation track, we have several categories of generators, random, constructive and genetic generators [Khalifa *et al.*, 2017]. Again, we worked on and improved the constructive and genetic generators. Since the methods are somewhat similar, we only discuss the most important differences and additions.

Constructive Generator

The constructive rule generator is more complicated than its level generator counterpart. In order to generate playable

games, it uses a template based approach. The template itself is based on some basic game knowledge, like for example killing the avatar when it collides with an enemy sprite.

Genetic Generator

We again use the FI2Pop algorithm [Kimbrough *et al.*, 2008] for the genetic generator, where we have the infeasible and feasible population. We made the same improvements to the genetic generator as in the level generation track. Moreover, there are three different mutation operators defined, insertion, deletion and modification. They respectively insert, delete or modify a game rule.

3.3 Learning Agents

The learning part involves using the generations from the previous subsections to train existing reinforcement learning agents. These agents come from algorithms in the Stable Baselines package which is based on OpenAI's own Baselines package. The reason we chose to use the Stable Baselines package instead of the regular Baselines package is that they claim to have improved implementations of the same algorithms, and they offer documentation which Baselines is lacking. Thanks to [Torrado *et al.*, 2018], the GVGAI games are ported to OpenAI Gym environments, which makes it possible to handle them as any other Gym environment and use these existing algorithms. This means our generated games are easily ported as well.

3.4 Experiment

The first part of our experiment involves generating our own games with the algorithms described earlier. We decided that we would stick to three games (Aliens, Boulderdash and Zelda) and that we would generate 4 levels with a mixture of rules for each of them. This way we created a training set of 12 levels. To decide which levels we would use, we got a little help which we explain later in the Section 4.

The three reinforcement learning algorithms we chose to use are A2C, TRPO and PPO2. The PPO2 algorithm combines ideas from A2C and TRPO and is known to be the best performance wise. This makes them interesting to compare as we should expect PPO2 to outperform the other two.

Each of the agents was trained for 10000 timesteps on each of the games. This means every agent was trained for a total of 120000 timesteps. Table 1 gives an overview of this to clarify (note that each of the games has 4 generated levels, thus equalling 40000 timesteps). The GVGAI competition allows you to upload your agents and run it on their servers. However, this was currently unavailable and thus we were limited to our own laptops, which is why we had to keep the experiment relatively small scale. The good thing was that the Stable Baselines package allows for continual learning which is ideal for this case as we are training general agents that can play multiple games and not just one.

Section 4 will discuss the comparison of the agents trained on 4 of our own generated levels to other agents (with the same algorithms) trained on 4 example levels provided by the GVGAI framework, by letting them play the 5th example level and noting down the scores for each run.

	Algorithms		
Games	A2C	TRPO	PPO2
Aliens	40000	40000	40000
Boulderdash	40000	40000	40000
Zelda	40000	40000	40000

Table 1: Algorithms and the games with the timesteps

4 Results

Evaluating generated levels and rules is not an easy task since there exist no automated way to evaluate them, it is a rather subjective process. We first look at the level generation methods.

4.1 Level Generation

Since we worked on two different generator types, the constructive and genetic generator, we first want to compare which performs better and is favoured by the game players. We have, therefore, performed a study where we asked ten human players to evaluate our generated games. We generated levels for the games of Zelda, Aliens, Boulderdash and Bomberman using both the constructive and genetic generator. We then asked the players whether they preferred the games generated by the constructive or genetic generator. The results are shown in Table 2. It shows that most players prefer the improved genetic generator over the improved constructive generator.

Looking further at why this might be the case, we found that the improved constructive generator does not always generate playable levels. In the fourth step of our improved method we fill ten percent of the level with random sprites, there is no post-processing step which ensures that the termination condition of the game can be reached. This fourth step of the algorithm also explains why some players reported the game of Bomberman to be 'strange' and felt like it was empty since only ten percent was filled by the remaining sprites. Furthermore, when we looked at all levels generated by these generators we discovered that some were either not solvable, e.g. a missing key in Zelda, or they were easy. This all shows that it is hard to design a generator which works well on all games.

	Constructive	Genetic
# of players preferring generator	3	7

Table 2: Amount of human players preferring each type (constructive versus genetic) of the level generator

Not only did we compare our two improved generators with each other, we also asked the ten human players to rate

whether they preferred levels generated by the standard version or the improved version of the generator. When choosing whether they preferred a level generated by one generator over the other the participants of the study had to take the following metrics into account, how playable the level was, how much they enjoyed the level, and if it had the right difficulty (not too easy neither too hard). The results are shown in Table 3. Looking at the constructive generator, both the standard and improved version performed equally well. When we consider the genetic generator, on the other hand, we see that more players preferred our improved version over the standard version. When asked for the reason, the players said they enjoyed the levels by the improved version more than the ones generated by the standard version.

# of players preferring	Constructive	Genetic
the standard version	5	4
the improved version	5	6

Table 3: Amount of human players preferring each version (standard versus improved) of the level generator

Looking at its generating performance, the improved constructive generator is faster than its genetic counterpart, although there is no assurance that the level being generated can be solved. The improved genetic generator takes longer since it has to work through several generations of levels. Moreover, its fitness function is influenced by an automatic planner/learner agent playing the generated levels.

4.2 Rule Generation

Similarly to the level generation track, we did a study where we asked the same ten human players to compare the constructive and genetic rule generators with regards to preference and playability. Table 4 shows the results. Interestingly, the constructive generator was preferred over the genetic generator. The reason for this probably lies in the use of a template-based approach as mentioned in Section 3.2, which is used to create well playable games by utilizing some basic game knowledge.

	Constructive	Genetic
# of players preferring generator	6	4

Table 4: Amount of human players preferring each type (constructive versus genetic) of the rule generator

We also looked at how the improved version compare with their standard counterparts. For this we asked the ten human players to rate whether they preferred the standard or improved version. Just as with the level generators, the players had to take several parameters into account when rating, these were, how playable the games were, how much they enjoyed the games with the generated rules, and if they were well-balanced in terms of difficulty. The results are shown in Table 5. We see that more players preferred the standard version of the constructive generator, while for the genetic generator the improved version was clearly favoured over the standard version.

# of players preferring	Constructive	Genetic
the standard version	6	3
the improved version	4	7

Table 5: Amount of human players preferring each version (standard versus improved) of the rule generator

Similar to the level generation track, the constructive generator is faster than the genetic generator in terms of producing results (rules for games). But again, this does not mean that the constructive generator works better or is preferred as we just showed above.

4.3 Learning Track

The learning part of the experiment did not go as smoothly as expected. There were lots of small hiccups on the way trying to get the algorithms to learn properly. Even though there is some documentation on the Stable Baselines package, there is barely any for the OpenAI Gym version of GVGAI. This made it harder to get everything setup properly, even though the code itself is quite basic. After we finally got it to work, it looked like the agents were not learning at all. For example, in a level of the game of Aliens, the agent stayed in the same spot and just shot bullets and did not move at all. This seemed strange to us at first, but we realized later that we should not expect an agent to play the game like we would ourselves and that it was actually a viable way of playing, as he would still win the game by doing just that.

The fact that we were limited to our laptops also did not help. Of course more processing power would make it easier. We had multiple nights where we would make the algorithms learn all night, to only wake up to a new error. The lack of proper feedback during learning also did not make it easier.

The results for the agents trained on our own generated levels are given by Table 6 and the results for the agents trained on the example levels are given by Table 7. The results unfortunately are not really telling us much. This was expected as we were very limited in our training. If we compare our experiment to that of [Ruben Rodriguez Torrado, 2018], ours is of a too small scale to really start comparing algorithms. Their experiment involved a million timesteps (while ours is only about $1/10th$ of that) and they only started to see big improvements at approximately 800000 timesteps for Boulderdash and the A2C algorithm for example.
At this point, without any further training, we can not really compare whether our levels would train the agent as well as the example levels. We also realise that we should repeat the same algorithm on a game more than 5 times to get a more realistic comparison.

5 Conclusion

Generating levels and rules for one specific game is quite an easy task since knowledge about the game can be used as an input to the generator. In our study, however, this is different. We are not dealing with generators for just one game, but rather a range of games. The generators should be able to

	Algorithms		
Games played	A2C	TRPO	PPO2
Aliens 1	51	53	51
Aliens 2	9	51	52
Aliens 3	54	24	48
Aliens 4	17	51	54
Aliens 5	52	62	16
Boulderdash 1	0	0	0
Boulderdash 2	2	0	0
Boulderdash 3	0	2	8
Boulderdash 4	6	0	2
Boulderdash 5	0	0	0
Zelda 1	0	0	1
Zelda 2	4	5	0
Zelda 3	0	1	0
Zelda 4	0	0	3
Zelda 5	0	0	0

Table 6: The scores for the agents trained on our levels playing the example levels 5 times each

	Algorithms		
Games played	A2C	TRPO	PPO2
Aliens 1	47	51	28
Aliens 2	51	55	57
Aliens 3	36	37	56
Aliens 4	55	50	17
Aliens 5	52	40	54
Boulderdash 1	0	6	2
Boulderdash 2	4	2	0
Boulderdash 3	2	4	0
Boulderdash 4	2	0	0
Boulderdash 5	2	2	2
Zelda 1	2	2	1
Zelda 2	4	4	0
Zelda 3	3	0	2
Zelda 4	0	0	0
Zelda 5	0	0	0

Table 7: The scores for the agents trained on the example levels playing the example levels 5 times each

generate levels and rules for any game. It is, therefore, not possible to use any existing knowledge about the game as an input to the generator. Generalizing the generators is not an easy task as we showed in Section 4. Some generators will produce very good results for one kind of game, while it might generate unsolvable games for another kind. We showed that the improved genetic generator is preferred for generating levels as it generated more playable levels, while for generating rules the improved constructive method using a template-based approach is preferred. We also showed that, especially in the level generation track, our improved version were preferred over the standard version of the generators. In the rule generation track, on the other hand, the standard constructive generator performed well as it is already quite advanced by using the template-based approach. Nonetheless, we showed that generating levels and rules that work well on all games is a hard task. As such, there are many opportunities for improvement and research in the field.

With regards to learning, we felt that it was unnecessarily 'hard' to do as there is a lack of proper documentation and tutorials, which means that if you get an error, you are basically stuck. We had to do a lot of trial-and-error to get it working and we feel like we still did not properly execute the experiment like we wanted to. However, for the scope of this course we believe we did a decent job.

This field is also still very open for further research. The GVGAI competition in 2020 is only available for the Learning track and can be found at http://www.aingames.cn/gvgai/ppsn_cog2020.

6 Time Spent

We tracked the time spent on the project using the mobile application Toggl [2]. In total we tracked 112 hours and 32 minutes for both of us, this means we each spent 56 hours and 16 minutes on average on the project. Besides that, running the learning agents also took many hours/days.

References

[Ahmed Khalifa and Togelius, 2016] Simon Lucas Ahmed Khalifa, Diego Perez-Liebana and Julian Togelius. General video game level generation. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), 2016*. IEEE, 2016.

[Ahmed Khalifa, 2017] Diego Perez-Liebana Simon Julian Togelius Ahmed Khalifa, Michael Green. General video game rule generation. In *IEEE Computational Intelligence and Games (CIG), 2017*. IEEE, 2017.

[Diego Perez, 2015] Julian Togelius-Tom Schaul Simon M. Lucas-Adrien Couetoux Jerry Lee Chong-U Lim Tommy Thompson Diego Perez, Spyridon Samothrakis. The 2014 general video game playing competition. In *IEEE Transactions on Computational Intelligence and AI in Games (CIG), 2015*. IEEE, 2015.

[Drageset *et al.*, 2019] O. Drageset, M. H. M. Winands, R. D. Gaina, and D. Perez-Liebana. Optimising level generators for general video game ai. In *2019 IEEE Conference on Games (CoG)*, pages 1–8, 2019.

[Ebner *et al.*, 2013] Marc Ebner, John Levine, Simon M. Lucas, Tom Schaul, Tommy Thompson, and Julian Togelius. Towards a Video Game Description Language. In Simon M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, and Julian Togelius, editors, *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-Ups*, pages 85–100. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2013.

[Khalifa *et al.*, 2017] A. Khalifa, M. C. Green, D. Perez-Liebana, and J. Togelius. General video game rule generation. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 170–177, 2017.

[2] https://toggl.com/

[Kimbrough *et al.*, 2008] Steven Orla Kimbrough, Gary J. Koehler, Ming Lu, and David Harlan 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*, 190(2):310 – 327, 2008.

[Ruben Rodriguez Torrado, 2018] Julian Togelius Jialin Liu Diego Perez-Liebana Ruben Rodriguez Torrado, Philip Bontrager. Deep reinforcement learning for general video game ai. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2018.

[Torrado *et al.*, 2018] Ruben Rodriguez Torrado, Philip Bontrager, Julian Togelius, Jialin Liu, and Diego Perez-Liebana. Deep reinforcement learning for general video game ai. In *Computational Intelligence and Games (CIG), 2018 IEEE Conference on*. IEEE, 2018.