An Application of ASP for Procedural Content Generation in Video Games

Andrea De Seta, Mario Alviano

DEMACS, University of Calabria, Via Bucci 30/B, 87036 Rende (CS), Italy

Abstract

Procedural content generation eases and accelerates the development of video games by creating data algorithmically through a combination of human-generated assets and algorithms usually coupled with computer-generated randomness. This paper presents a use case of Answer Set Programming (ASP) for procedural content generation of levels in a rougelike video game powered by the Godot Engine. The main elements of a set of human-generated rooms are represented by ASP facts, among them positions of doors, presence of treasures and power-ups. Within this knowledge, ASP is asked to generate dungeons satisfying a few conditions, among them the correct positioning of rooms, the absence of unreachable rooms and constraints on the occurrences of rooms. Scalability of ASP in this context is evaluated empirically, showing that it can generate in few seconds levels that comprise thousands of rooms.

1. Introduction

Video game industry, i.e. the industry involved in the development, marketing and monetization of video games, has grown sensibly in the recent years, and its annually generated sales are in the order of hundreds of billions of dollars. The development of a video game is the first step to enter such an industry, and several frameworks are nowadays available to start with a set of common primitives that can be combined to achieve appealing results in relatively short time. Among them there is the Godot Engine (https://godotengine.org/), a completely free and open-source game engine under MIT license, providing a huge set of common tools that, especially for 2D-games, significantly accelerate all the development phase.

Further acceleration in the development of video games can be provided by techniques that go under the name of *procedural content generation (PCG)*, and essentially consist of algorithms that automatically create levels, maps, weapons, background scenery, and music for video games [1, 2]. The idea is not new, and actually exploited already in the '70s in historical titles such as PEDIT5 (https://en.wikipedia.org/wiki/Pedit5) to circumvent the limited amount of computational resources [3]. Many video games followed the idea of PEDIT5 and took advantage of PCG. Later, those video games were classified as *rougelike*, a term originated from '90s USENET newsgroups. Even if the exact definition of a rougelike game remains a point of debate in the video game community, some characteristics are clearly identified. Specifically, rougelike is a subgenre of role-playing video games characterized by a dungeon crawl through PCG levels, turn-based gameplay, grid-based movement, and permanent death of the player character.

CILC 2022: 37th Italian Conference on Computational Logic, June 29 – July 1, 2022, Bologna, Italy

In this work the term rougelike is slightly abused and used to refer to video games that have all the aforementioned characteristics but turn-based gameplay and grid-based movement. Stated differently, in the following we consider a video game characterized by a dungeon crawl through PCG levels, *real-time gameplay*, *grid-based positioning but free movements*, and permanent death of the player character. The presented game is entitled WizardSet, is powered by the Godot Engine and takes advantage of Answer Set Programming (ASP; [4, 5, 6]) for PCG of levels. In a nutshell, the idea underlying WizardSet is to design some elements of the video game with the Godot Engine, represent such elements in a format understandable by ASP systems, and then combine such a representation with an ASP encoding to delegate the generation of a complete level to an ASP reasoner. This way, levels are generated when needed, providing a unique experience to the player for each game. After introducing the required background knowledge (Section 2), we will detail on the PCG implemented in WizardSet (Section 3), and report on the result of an experiment aimed at assessing the scalability of the proposed procedure (Section 4).

2. Background

The Godot Engine provides an Integrated Development Environment to design scenes, where a scene represents an element of the video game and is characterized by a tree and possibly a script. The tree of a scene is made of nodes of different nature that can be used to associate sprites and physical properties to the element of the video game. Scripts are usually written in GDScript, a high-level, dynamically typed programming language with a syntax similar to Python (but a few other languages are supported as well).

ASP is a rule-based language supporting object variables and negation under stable model semantics. Object variables are removed by means of intelligent grounding, and stable models are searched by conflict-driven clause learning algorithms. ASP systems extend the basic language of ASP with several constructs for representing common knowledge, among them integer arithmetic and aggregates. The reader is referred to the ASP Core 2 [7] format for details.

3. Procedural Content Generation within ASP

WizardSet represents a few key features of its rooms in terms of ASP facts:

- `room(`r`)`, for each available room, where r is a positive integer identifying the room (the current release of WizardSet provides 17 rooms with different layout and features);
- `room_door(`r`,`l`)`, if room r has a door in the wall l, where l is one of `north`, `south`, `west`, and `east`;
- `room_flag(`r`,`f`)`, if room r has feature f, where f is among `initial`, `boss`, and `treasure`;
- `flag_bounds(`f`,`min`,`max`)`, for each represented feature, where min and max are integers to bound the number of rooms with feature f in a level; in particular, we fix `flag_bounds(initial,1,1)`, `flag_bounds(boss,1,1)`, and `flag_bounds(treasure,0,1)`, i.e. there must be exactly one initial room and boss room, and at most one treasure room in each generated level.

Figure 1: Design view of room 4 of WizardSet, having a door in the east wall and containing a treasure. The room is represented by the following ASP facts: `room(4)`, `room_door(4,east)`, and `room_flag(4,treasure)`.

An example room and the associated ASP facts are reported in Figure 1. Additionally, the ASP encoding is enriched with the following facts to represent movements in the four cardinal directions:

```
delta(north,-1, 0).     opposite(north,south).
delta(south, 1, 0).     opposite(south,north).
delta(west,  0,-1).     opposite(west, east).
delta(east,  0, 1).     opposite(east, west).
```

The reasoning core of the ASP encoding empowering WizardSet is parameterized by the size of the grid to generate (constants `rows` and `cols`), the number of rooms to deploy (constant `required_rooms`), and the health points of the player (constant `hp`). The ASP encoding non-deterministically assigns rooms to cells of the grid, ensuring that a few constraints are satisfied. The following ASP rules are used (and described below):

```
r1 :  {assign(X,Y,nil); assign(X,Y,R) : room(R)} = 1 :-
          X = 1..rows, Y = 1..cols.
r2 :  assign(0,     Y,     nil) :- Y = 0..cols+1.
r3 :  assign(rows+1,Y,     nil) :- Y = 0..cols+1.
r4 :  assign(X,     0,     nil) :- X = 0..rows+1.
r5 :  assign(X,     cols+1,nil) :- X = 0..rows+1.
r6 :  :- #count{X,Y : assign(X,Y,R), R != nil} != required_rooms.
r7 :  :- assign(X,Y,R), room_door(R,L), delta(L,DX,DY),
```

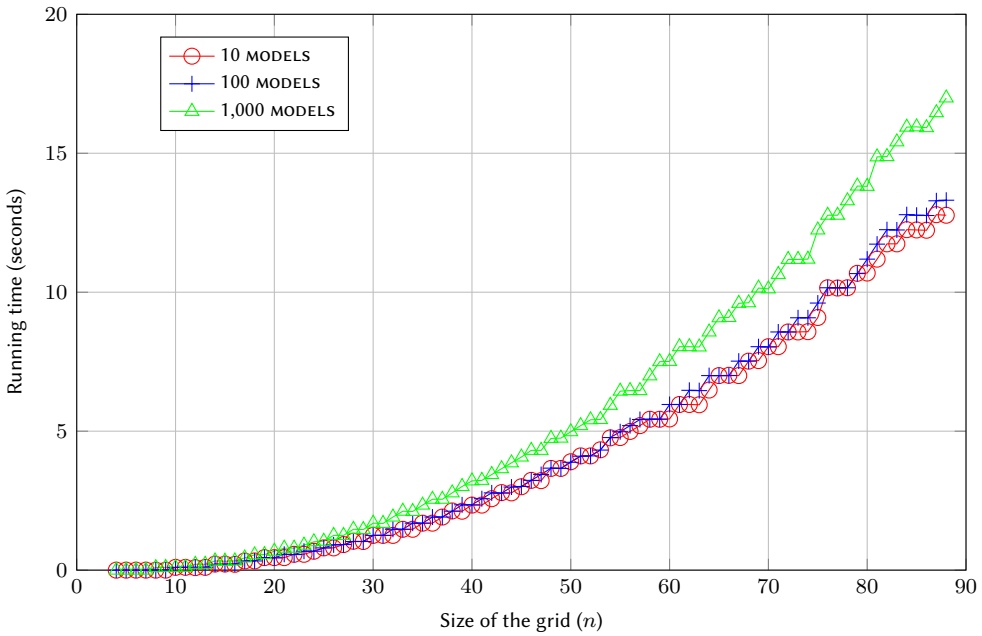

Figure 2: CLINGO performance on $n \times n$ level generation

```
          assign(X+DX,Y+DY,R'), opposite(L,L'), not room_door(R',L').
r8 :  :- flag_bounds(F,MIN,MAX),
          not MIN <= #count{X,Y : assign(X,Y,R), room_flag(R,F)} <= MAX.
r9 :  reachable(X,Y) :- room_flag(R,initial), assign(X,Y,R).
r10 : reachable(X+DX,Y+DY) :- reachable(X,Y), assign(X,Y,R),
                              room_door(R,L), delta(L,DX,DY).
r11 : :- assign(X,Y,R), R != nil, not reachable(X,Y).
r12 : spawn_hp_potion(R) :- room_flag(R,initial), hp <= 2.
```

The assignment itself is generated by rule r_1, which associates each cell with a room or with the `nil` value (to denote a non-playable cell of the grid, i.e. a cell not containing any room). Rules r_2–r_5 introduce a border of non-playable cells, so to clearly delimit the playable boundary of the generated level. Rule r_6 enforces that the number of assigned rooms is the required one, and rule r_7 ensures the correct placement of doors. Rule r_8 ensures that the presence of represented features in the required amount. Rules r_9–r_{11} impose that all playable cells are actually connected. Finally, rule r_{12} provides an example of power-up that can be placed in some rooms of the generated dungeon; in this case, the power-up is a health potion to be placed in the initial room if the player starts with few health points.

4. Implementation and Experiment

WIZARDSET is open-source (https://github.com/Tatanka4/WizardSet). The ASP system adopted for PCG is CLINGO 5.5.0 [8], and communication between the Godot

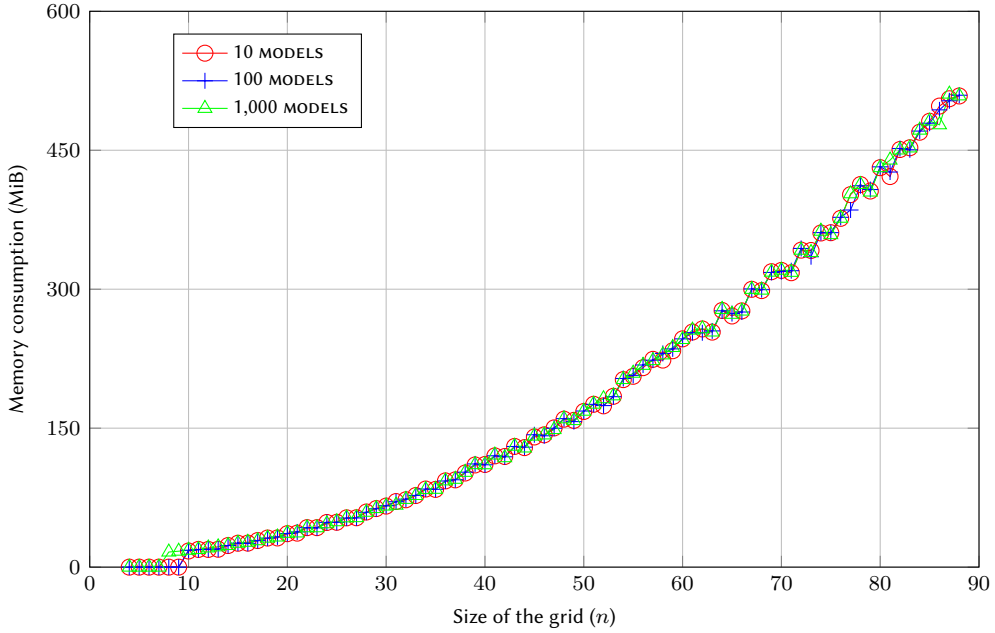

Figure 3: CLINGO memory consumption on $n \times n$ level generation

Engine and CLINGO is achieved by synchronous system calls and file sharing. Parameters for the generation of levels are adjusted to increase the size of the grid and the number of playable cells, and therefore the expected difficulty. On ordinary playing sessions the time required for PCG is negligible, and usually below a second. Therefore, in order to empirically evaluate the scalability of the proposed ASP encoding, we designed a first experiment to measure the execution time required by CLINGO to generate squared grids of increasing size $n \times n$, with the number of requested rooms set to $\frac{n \times n}{2}$. All tests are run on an Intel Xeon 2.4 GHz with 16 GiB of memory. Time and memory were limited to 20 seconds and 4 GiB (WIZARDSET is expected to be run on laptops and low-end PCs, and level generation must stay in the order of a few seconds).

Results are shown in Figure 2 for different settings of the number of models asked to CLINGO; in fact, to increase randomness in the generation of levels, WIZARDSET selects one model among several that are produced by CLINGO. Additionally, we observed that the generation of levels is unfeasible without limiting the number of models; the reason is to be attributed to the high number of possible models, which is already 2,064 for grids of size 4×4 (0.1 seconds of computation) and 1,360,822 for grids of size 5×5 (89 seconds of computation). We therefore ran all test cases by limiting the number of models to 10, 100 and 1,000. Within these limits, levels can be generated in less than 20 seconds up to grids of size 89×89. Figure 3 reports the memory consumption for producing 10, 100 and 1,000 models, and it can be observed that almost the same consumption was measured. Given the results of our experiment, in the release of WIZARDSET we fixed the number of models to ask to CLINGO to 1,000 models, as there is no significant performance gain in asking for less models.

5. Conclusion

Previous works in the literature have already shown applications of ASP to video games. For example, ASP is used in Angry-HEX [9], an AI that can play Angry Birds, and in ThinkEngine [10, 11], an integration of ASP in Unity. ASP can be used profitably for PCG in video games as well, and the idea was already used in the literature [12, 13, 14]. This work provides another concrete example of ASP-powered PCG by presenting the first combination of the Godot Engine with an ASP system to generate levels of a rougelike video game. In our experience, the main advantage of adopting ASP for this task relies in the declarative power of the language, thanks to which constraints and desiderata for PCG can be specified succinctly in a few lines of code. A similar observation is reported in [14], whose approach to generate dungeons is to partition the space in rectangular areas, and then generate a random room in each area; in this case, the ASP encoding models desiderata on the generation of rooms. In our approach, rooms are defined by the programmer, their features represented in ASP, and such a knowledge base is used to generate a dungeon. Moreover, our ASP representation is suitable for future extensions of the approach, as for example by enriching the knowledge base with other features of the rooms so to control the number of enemies and power-ups in the generated levels.

Acknowledgments

This work was partially supported by the projects PON-MISE MAP4ID (CUP B21B19000650008) and PON-MISE S2BDW (CUP B28I17000250008), by the LAIA lab (part of the SILA labs) and by GNCS-INdAM.

References

[1] M. Hendrikx, S. A. Meijer, J. V. D. Velden, A. Iosup, Procedural content generation for games: A survey, ACM Trans. Multim. Comput. Commun. Appl. 9 (2013) 1:1–1:22. doi:10.1145/2422956.2422957.

[2] J. Togelius, G. N. Yannakakis, K. O. Stanley, C. Browne, Search-based procedural content generation: A taxonomy and survey, IEEE Trans. Comput. Intell. AI Games 3 (2011) 172–186. doi:10.1109/TCIAIG.2011.2148116.

[3] N. Brewer, Computerized dungeons and randomly generated worlds: From rogue to minecraft, Proc. IEEE 105 (2017) 970–977. doi:10.1109/JPROC.2017.2684358.

[4] M. Gelfond, V. Lifschitz, Classical negation in logic programs and disjunctive databases, New Generation Computing 9 (1991) 365–386. doi:10.1007/BF03037169.

[5] V. W. Marek, M. Truszczyński, Stable models and an alternative logic programming paradigm, in: The Logic Programming Paradigm – A 25-Year Perspective, Springer Verlag, 1999, pp. 375–398. doi:10.1007/978-3-642-60085-2_17.

[6] I. Niemelä, Logic programming with stable model semantics as constraint programming paradigm, Annals of Mathematics and Artificial Intelligence 25 (1999) 241–273. doi:10.1023/A:1018930122475.

[7] F. Calimeri, et al, Asp-core-2 input language format, Theory Pract. Log. Program. 20 (2020) 294–309. doi:10.1017/S1471068419000450.

[8] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, P. Wanko, Theory solving made easy with clingo 5, in: M. Carro, A. King (Eds.), TC of ICLP'16, volume 52 of *OASIcs*, Schloss Dagstuhl, Germany, 2016, pp. 2:1–2:15.

[9] F. Calimeri, et al., Angry-hex: An artificial player for angry birds based on declarative knowledge bases, IEEE Trans. Comput. Intell. AI Games 8 (2016) 128–139. doi:10.1109/TCIAIG.2015.2509600.

[10] D. Angilica, G. Ianni, F. Pacenza, Tight integration of rule-based tools in game development, in: M. Alviano, G. Greco, F. Scarcello (Eds.), AI*IA 2019, Rende, Italy, November 19-22, 2019, Proceedings, volume 11946 of *LNCS*, Springer, 2019, pp. 3–17. doi:10.1007/978-3-030-35166-3_1.

[11] F. Calimeri, S. Germano, G. Ianni, F. Pacenza, S. Perri, J. Zangari, Integrating rule-based AI tools into mainstream game development, in: C. Benzmüller, F. Ricca, X. Parent, D. Roman (Eds.), RuleML+RR 2018, Luxembourg, September 18-21, 2018, volume 11092 of *LNCS*, Springer, 2018, pp. 310–317. doi:10.1007/978-3-319-99906-7_23.

[12] X. Neufeld, S. Mostaghim, D. Perez Liebana, Procedural level generation with answer set programming for general video game playing, 2015, pp. 207–212. doi:10.1109/CEEC.2015.7332726.

[13] A. M. Smith, M. Mateas, Answer set programming for procedural content generation: A design space approach, IEEE Trans. Comput. Intell. AI Games 3 (2011) 187–200. doi:10.1109/TCIAIG.2011.2158545.

[14] F. Calimeri, S. Germano, G. Ianni, F. Pacenza, A. Pezzimenti, A. Tucci, Answer set programming for declarative content specification: A scalable partitioning-based approach, in: AI*IA, volume 11298 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 225–237.