Compositional procedural content generation

Julian Togelius
IT University of Copenhagen
Rued Langgaards Vej 7
2300 Copenhagen, Denmark
julian@togelius.com

Tróndur Justinussen
IT University of Copenhagen
Rued Langgaards Vej 7
2300 Copenhagen, Denmark
tjus@itu.dk

Anders Hartzen
IT University of Copenhagen
Rued Langgaards Vej 7
2300 Copenhagen, Denmark
andershh@itu.dk

ABSTRACT

We consider the strengths and drawbacks of various procedural content generation methods, and how they could be combined to hybrid methods that retain the advantages and avoid the disadvantages of their constituent methods. One answer is composition, where one method is nestled inside another. As an example, we present a hybrid evolutionary-ASP dungeon generator.

1. INTRODUCTION

A procedural content generation (PCG) problem takes the following form: for a given game, player, context and content type, generate one or several content artefacts that are good enough according to some criterion. For example, one might want to generate a map that allows each player a starting position with surrounding land to explore (Civilization), a set of weapons that are interestingly different from each other but not excessively powerful (Borderlands), or simply plants that are diverse and natural-looking (SpeedTree). PCG problems can be posed and solved online as the game is executing, or offline during design time.

Like all problems, PCG problems can be cast as search problems. From this perspective, solving a problem means searching for a solution to the problem. There might exist null, one or many (possibly very many) solutions that are good enough. In a recent survey paper [7], various ways in which PCG can be seen as search are discussed and some of the major choices in searching for good content are outlined, including representation, evaluation function and search algorithm. The vast majority of work cited in that survey uses evolutionary computation or similar global stochastic search algorithms.

An important exception to this is the use of Answer Set Programming (ASP) for PCG. As introduced by Smith and Mateas [5, 4], this technique uses the AnsProlog language to encode content artefacts and the validity constraints such content must adhere to. While syntactically similar to Prolog, AnsProlog is interpreted in a very different way: running an AnsProlog program returns the set of all configurations of the specified variables that are valid "answers" to the program, i.e. all configurations of variables that do not contradict the validity constraints. When using ASP for PCG, each answer generated can be interpreted as an artefact.

ASP does not completely fit the definition of "search-based" PCG in [7], as most ASP implementations do not use stochastic search. Most ASP solvers implement some deterministic process that iteratively prunes search space through elimination of inconsistent configuration, and which does not generate and test candidate answers. (A common technique is to translate the answer set program into an instance of the satisfiability problem, and use a SAT solver.) However, content generation based on ASP undoubtedly performs a search of the content space. The name "solver-based PCG" has been proposed for ASP and similar techniques (such as constraint solving as used in e.g. Tanagra [6]), but one could just as well choose to widen the definition of "search-based" to include these approaches.

The very different modes of search and problem specification employed by evolution-like algorithms and by ASP have their respective advantages and disadvantages. A complete characterisation of these differences is still outstanding and certainly beyond the scope of this paper. However, one difference that can be pointed out, is that whereas an execution of an ASP program terminates after an unknown time with a set of unknown size (perhaps zero) of artefacts that are guaranteed to satisfy all of the constraints that have been specified, a run of an evolutionary algorithm terminates after a well-known pre-specified time with a set of well-known pre-specified size of artefacts that satisfy all of the objectives that have been specified as best as the algorithm could find (perhaps not very well at all). While in practice a well-specified ASP program terminates very quickly, the worst-case complexity is exponential. On the other hand, while in practice a well-tailored representation and objective function allows an evolutionary algorithm to find good solutions quickly, there is certainly no guarantee.

All search-based methods depend on that the desirable qualities of content can somehow be encoded as evaluation functions (objectives) or constraints (any evaluation function can be turned into a constraint by specifying a minimum acceptable value). These functions should reliably and accurately measure what they are intended to measure, and efficiently

computable. Finding good evaluation functions is a major problem for all PCG approaches that rely on search. It seems extremely hard to design an algorithm that accurately measures something as complex as how interesting a game ruleset is. Therefore some approaches measure a number of simpler features and try to combine these measures, or alternatively use humans as part of the loop to evaluate candidate content artefacts.

Many other PCG techniques, however, do not perform any search of content space, at least not in the same sense as evolutionary algorithms or ASP search space. Classic PCG algorithms such as Diamond-square [1], Perlin noise [3] or cellular automata [2] are *constructive*: they run once and produce a result within a bounded and predictable (and usually short) runtime, without the need for re-generating content in case the generation somehow went wrong. While there is much to be said for short and predictable runtimes, it is hard to create constructive algorithms for many types of content that are capable of generating a sufficiently rich and diverse repertoire of content without risk of catastrophic failure, i.e. generating content that is so bad in some sense that it breaks the game. This is probably why constructive PCG methods have previously only been used for optional content (which can be broken without destroying gameplay) or exhaustively tested to never produce broken content (as in the case of *Elite*).

2. HYBRIDISATION THROUGH COMPOSITION

So it seems we have a number of different approaches to PCG, none of which is a silver bullet: they all have advantages and disadvantages. This should not come as a great surprise to anyone. The precise merits and problems of individual methods have been debated before and keep being debated, but that is not our focus here. Instead we want to discuss how different PCG methods can be combined into *hybrid* methods, which combines the strengths of several methods while avoiding the weaknesses of either.

One way of combining methods is to unite them within a search-based framework. A key part of any search-based content generation method is the genotype-to-phenotype mapping. This mapping is whatever transforms the genotype (the data structures used inside the search algorithm) into the phenotype (the actual content artefacts). For example, transforming vectors of real numbers into complete RTS maps, or transforming grammar axioms into 3D rocks. As observed in [7], the genotype-to-phenotype mapping can be seen as a PCG algorithm in its own right, that takes the genotype as input (parameters) and produces the phenotype as output. The more indirect the mapping between genotype and phenotype, the more is expected of this algorithm. In order for the search to be effective, it is important that the mapping preserves *locality*, i.e. that small changes in genotype space produce proportionally small changes in phenotype space.

The mapping (or "inner PCG algorithm") could consist of either a constructive, generate-and-test, solver-based or another search-based algorithm. Given that the different approaches to PCG seem to excel at producing (or safeguarding) different qualities/properties in content artefacts, it would make sense to choose a mapping that is complementary to the main ("outer") PCG algorithm in terms of what properties it is responsible for.

Figure 1 depicts several desirable properties common to many types of game content, ordered by ascending difficulty. (Any likeness to classic concepts and diagrams in psychology is completely coincidental.) At the bottom of the pyramid we find those basic aspects of game content which need to be present in order for the content to make sense at all — it needs to be consistent (objects have finite dimensions, rulesets are non-contradictory) and playable (maps have starting positions and game boards have free space). There should also exist a winning condition and be possible to get there from the starting condition. Constructive algorithms are usually good at producing consistent content, and playability and winnability can often be specified as easily checked and satisfied constraints, and thus be delivered by a solver-based PCG algorithm. Further up the ladder we find such properties as challenge and fairness (it is equally easy to play as white or blue even though the sides have different conditions, the game rewards you for playing better rather than worse). While such properties can often be reasonably approximated by simulation-based evaluation functions (playing part of the game with an artificial agent and evaluating based on the outcome), they are properly evaluated on a continuum rather than as constraints to be fulfilled. A game is not challenging or non-challenging, but it might be "0.76 challenging". Such numerical evaluation values are a natural fit for search-based methods. Finally, there are those properties for which we cannot currently find any good algorithmic approximations, but will simply have to put a human in the loop; properties such as "interesting", "creative", "awe-inspiring" and "sublime". At the top level, we will have to resort to mixed-initiative techniques or interactive evolution, but we should do everything we can to ensure that properties at lower levels are automatically optimised so that humans don't have to spend their scarce resources in ensuring these lower-level properties.

A good reason for combining several PCG algorithms, in our opinion, is to be able to produce content that has more desirable properties than could be delivered by any individual algorithm. A viable strategy for such combination of algorithms is *composition*: using an inner algorithm that guarantees the more basic properties within an outer algorithm that optimises the more advanced properties.

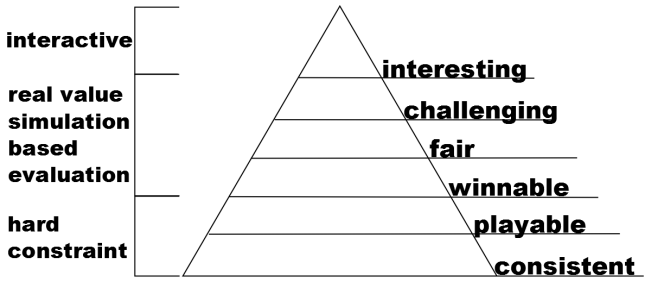

Figure 1: A pyramid of desirable properties, and how they could reasonably be evaluated.

3. COMPOSITIONAL DUNGEON GENERATION

As an example of hybrid content generation, we present an experiment where we create dungeons for a fictive roguelike game using an ASP-evolution hybrid approach. The genotypes are parameters for answer set programs, which are converted using an ASP solver to answer sets. Each answer is then interpreted as a map and evaluated using simulation-based testing; ASP is responsible for well-formedness, playability and winnability, whereas evolutionary search is responsible for optimising challenge and skill differentiation.

3.1 Representation and mapping

The chromosome consists of 17 discrete numbers specifying various controllable properties of generated dungeons:

- Height of the map
- Width of the map
- Minimum number of steps from start to finish
- X an Y coordinates of entrance and exit
- Number of traps
- Number of open tiles
- Number of monsters of types 1-4
- Number of "true sight" buffs (shows all traps in the map)
- Number of "disarm next trap" buffs
- Number of buffs that help in the next monster encounter

During genotype-to-phenotype mapping, these values are first transcribed into an answer set programming, by substituting placeholder values in an existing file (loosely based on Adam Smith's example maze generation code[1]). While we do not have space to reproduce the full ASP code here, the snippet that deals with specifying trap placement looks as follows (note that the word "trapcount" is replaced with the actual number of traps specified in the chromosome during the mapping process):

```
%Amount of traps
{trap(X,Y) :dim1(X) :dim2(Y)}trapcount.
%Traps should be reachable
:- trap(X,Y), not solid(X,Y).
:- trap(X,Y), not reachable(X,Y).
:- trap(X,Y), start(X,Y).
:- trap(X,Y), finish(X,Y).
```

This file is passed to an ASP solver, and the resulting answers are interpreted as dungeons. The AnsProlog program is written so that any dungeons that are generated this way are guaranteed to be not only complete and well-formed, but also winnable; there is at least one way to get from start to goal without dying.

[1]http://eis-blog.ucsc.edu/2011/10/map-generation-speedrun

3.2 Evaluation function

The evaluation function tries to measure the challenge and skill differentiation of the dungeon. With "skill differentiation" we mean that skilful playing should be rewarded with more in-game success than bad playing. (The opposite would be badly designed content where the player has little influence over what happens.) These properties are approximated through simulation-based evaluation; two different agents play the dungeon, and the difference between their results is measured. The reckless agent plays by simply following the shortest path (as found by A*) from start to goal, running straight into any monsters and traps along its way. The smart agent also relies on A*, but has a heuristic that causes it to avoid monsters and traps when possible and collect any buffs close to its path.

The evaluation function can be stated as

$$f = (1 - d_l/d_a) + (1 - h_s/h_a) \quad (1)$$

where d_l is the distance from start to finish in a straight line, d_a is the distance from start to finish along the shortest path (found with A*), h_s is the amount of damage sustained by the smart agent when navigating the dungeon and h_r is the amount of damage sustained by the reckless agent when navigating the dungeon.

Note that any particular genotype can produce zero or more phenotypes (answers). When the ASP solver returns the empty set as a result, indicating that no dungeons with the specified controllable properties exist, a fitness of zero is returned. Otherwise, the first 20 dungeons produced during the mapping are evaluated and their fitness averaged. In most cases, there is a relatively high fitness variance between the dungeons generated by the same ASP code.

3.3 Results

A $\mu+\lambda$ (elitist) evolution strategy (ES) with $\mu = 30, \lambda = 30$ was used as the core search algorithm. Mutation was performed with $p = 0.3$ for perturbing each locus. Several runs were made with a number of variations of the underlying ASP program, in order to find the variation that allowed the reliable production of the best dungeons. Figure 2 shows fitness growth during a typical evolutionary run. As can be seen the ES finds a fitness plateau after only about 40 generations in this case. Figure 3 shows an example generated dungeon, displayed in a format similar to that used in many roguelike games.

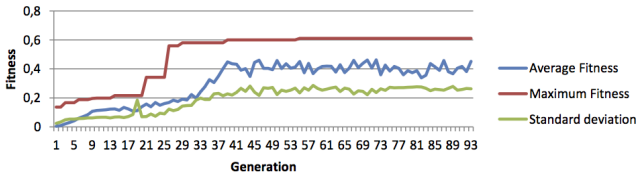

Figure 2: An evolutionary run of 92 generations.

4. CONCLUSIONS

We have discussed the pros and cons of some different families of PCG algorithms, and how they can be combined through "composition", or putting one algorithm inside another. In our discussion, this means using a constructive or

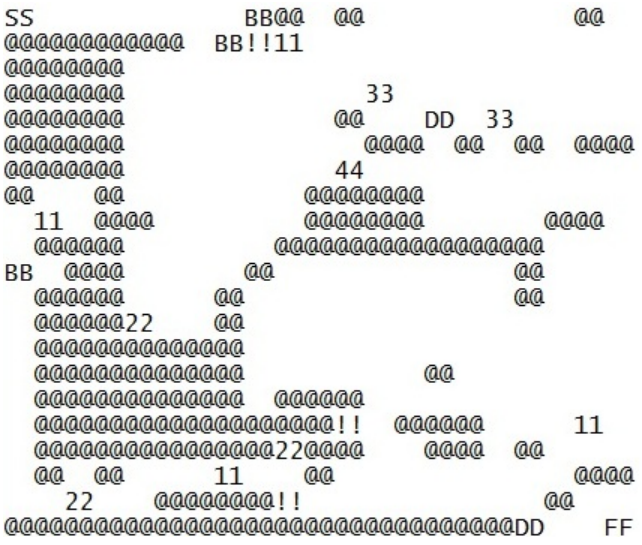

Figure 3: A sample generated map. Legend: @@–wall, !!–trap, 11–monster type 1 (etc for 2, 3, 4), TT–true sight buff, DD–disarm trap buff, BB–defeat monster buff, SS–start, FF–finish.

solver-based algorithm as the genotype-to-phenotype mapping of a search-based algorithm, though it is conceivable that composition could be achieved in some other manner as well. The advantage of doing this is that the "inner" algorithm could be used to guarantee the lower-level desirable properties that can be encoded as generation rules and/or constraints, leaving the "outer" algorithm to pursue the higher-level desirable properties which might require simulation-based testing or human input to approximate, and cannot be encoded as constraints due to there not being any known or easily attainable minimum level.

We also demonstrated compositional PCG with a simple experiment where dungeons for a roguelike game are generated with a hybrid approach. An evolution strategy evolves parameters for ASP programs, which when run produce answer sets, which in turn are interpreted as dungeons and evaluated using a simulation-based fitness function. While this is admittedly a simplistic example that does not use either algorithm to its full potential, we believe that the division of labour between the algorithms is the correct one, and in some more demanding content generation tasks such a division of labour might be essential to producing practical solutions.

5. ACKNOWLEDGEMENTS

Thanks to Adam Smith of UC Santa Cruz for inspiring discussions, and also for help with technical matters.

6. REFERENCES

[1] A. Fournier, D. Fussell, and L. Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, 1982.

[2] L. Johnson, G. N. Yannakakis, and J. Togelius. Cellular Automata for Real-time Generation of Infinite Cave Levels. In *Proceedings of the ACM Foundations of Digital Games*. ACM Press, June 2010.

[3] K. Perlin. An image synthesizer. In *ACM SIGGRAPH Computer Graphics*, volume 19, pages 287–296. ACM, 1985.

[4] A. Smith and M. Mateas. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 2011.

[5] 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*, 2010.

[6] G. Smith, J. Whitehead, and M. Mateas. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):201–215, 2011.

[7] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. Search-based procedural content generation: a taxonomy and survey. *IEEE Transactions on Computational Intelligence and Games*, 3:xx–xx, 2011.