Search-based Procedural Content Generation

Julian Togelius[1], Georgios N. Yannakakis[1],
Kenneth O. Stanley[2], Cameron Browne[3]

[1] IT University of Copenhagen, Rued Langaards Vej 7, 2300 Copenhagen, Denmark
[2] University of Central Florida, 4000 Central Florida Blvd. Orlando, Florida, 32816
[3] Imperial College London, London SW7 2AZ, UK
julian@togelius.com, yannakakis@itu.dk, kstanley@eecs.ucf.edu,
cameron.browne@btinternet.com

Abstract. Recently, a small number of papers have appeared in which the authors implement stochastic search algorithms, such as evolutionary computation, to generate game content, such as levels, rules and weapons. We propose a taxonomy of such approaches, centring on what sort of content is generated, how the content is represented, and how the quality of the content is evaluated. The relation between search-based and other types of procedural content generation is described, as are some of the main research challenges in this new field. The paper ends with some successful examples of this approach.

1 Introduction

In this paper we aim to define *search-based procedural content generation*, investigate what can and cannot be accomplished by the techniques that go under this name, and outline some of the main research challenges in the field. Some distinctions will be introduced between approaches, and a handful of examples of search-based procedural content generation (SBPCG) will be discussed within and classified according to these distinctions. It is important to note that this paper proposes an initial framework of SBPCG approaches that leaves room for further new approaches to be co-located within this young yet emerging field. To begin, procedural content generation is itself introduced.

Procedural content generation (PCG) refers to the creation of game content automatically, through algorithmic means. In this paper, *game content* means all aspects of a game that affect gameplay but are not non-player character (NPC) behaviour or the game engine itself. This definition includes such aspects as terrain, maps, levels, stories, dialogue, quests, characters, rulesets, camera viewpoint, dynamics and weapons. The definition explicitly excludes the most common application of search and optimisation techniques in academic games research, namely, NPC artificial intelligence.

There are several reasons for game developers to be interested in PCG. The first is memory consumption — procedurally represented content can typically be compressed by keeping it "unexpanded" until needed. A good example is the classic space trading and adventure game *Elite* (Acornsoft 1984), which managed to keep hundreds of star systems in the few tens of kilobytes of memory available on the hardware of the day by representing each planet as just a few numbers. Another reason for using PCG is the prohibitive expense of manually creating game content. Many current generation AAA titles employ software such as *SpeedTree* to create whole areas of vegetation based on just a few parameters, saving precious development resources while allowing large, open game worlds.

A third argument for PCG is that it might allow the emergence of completely new types of games, with game mechanics built around content generation. If new content can be generated with sufficient variety in real time then it may become possible to create truly endless games. Further, if this new content is created according to specific criteria, such as its suitability for the playing style of a particular player (or group/community of players) or based on particular types of player experience (challenge, novelty, etc.), it may become possible to create games with close to infinite replay value.

A fourth argument for PCG is that it augments our limited, human imagination. Off-line algorithms might create new rulesets, levels, narratives, etc., which can then inspire human designers and form the basis of their own creations.

2 Dissecting Procedural Content Generation

While PCG in different forms has been a feature of various games for a long time, there has not been an academic community devoted to its study. This situation is now changing with the recent establishment of a mailing list[4], an IEEE CIS Task Force[5], a workshop[6] and a wiki[7] on the topic. However, there is still no textbook on PCG, or even an overview paper offering a basic taxonomy of approaches. Therefore, this section aims to begin to draw some distinctions. Most of these distinctions are not binary, but rather a continuum wherein any particular example of PCG can be placed closer to one or the other extreme. Note that these distinctions are drawn for the purpose of clarifying the role of search-based PCG; of course other distinctions will be drawn in the future as the field matures.

2.1 Online versus Offline

The first distinction to be made is whether content generation is performed online during the runtime of the game, or offline during game development. An example of the former is when the player enters a door to a building and the game instantly generates the interior of the building, which was not there before; in the latter case an algorithm suggests interior layouts that are then edited and perfected by a human designer before the game is shipped. Intermediate cases are possible, wherein an algorithm running on e.g. an RTS server suggests new maps to a group of players daily based on logs of their recent playing styles.

[4] http://groups.google.com/proceduralcontent
[5] http://game.itu.dk/pcg/
[6] http://pcgames.fdg2010.org/
[7] http://pcg.wikidot.com

2.2 Necessary versus Optional Content

A further distinction relating to the generated content is whether that content is necessary or optional. Necessary content is required by the players to progress in the game — e.g. dungeons that need to be traversed, monsters that need to be slain, crucial game rules, and so on — whereas optional content is that which the player can choose to avoid, such as available weapons or houses that can be entered or ignored. The difference here is that necessary content always needs to be correct; e.g. it is not acceptable to generate an intractable dungeon if such an aberration makes it impossible for the player to progress. On the other hand, one can allow an algorithm that sometimes generates unusable weapons and unreasonable floor layouts if the player can choose to drop the weapon and pick another one or exit a strange building and go somewhere else instead.

2.3 Random Seeds versus Parameter Vectors

Another distinction concerning the generation algorithm itself is to what extent it can be parameterised. All PCG algorithms create "expanded" content of some sort based on a much more compact representation. At one extreme, the algorithm might simply take a seed to its random number generator as input; at another extreme, the algorithm might take as input a multidimensional vector of real-valued parameters that specify the properties of the content it generates.

2.4 Stochastic versus Deterministic Generation

A distinction only partly orthogonal to those outlined so far concerns the amount of randomness in content generation, as the variation in outcome between different runs of an algorithm with identical parameters is a design question. It is possible to conceive of deterministic generation algorithms that always produce the same content given the same parameters, but it is well known that many algorithms do not. (Note that we do not consider the random number generator seed a parameter here, as that would imply that all algorithms are deterministic.)

2.5 Constructive versus Generate-and-test

A final distinction may be made between algorithms that can be called *constructive* and those that can be described as *generate-and-test*. A constructive algorithm generates the content once, and is done with it; however, it needs to make sure that the content is correct or at least "good enough" as it is being constructed. An example of this approach is using fractals to generate terrains [1].

A generate-and-test algorithm incorporates both a generate and a test mechanism. After a candidate content instance is generated, it is tested according to some criteria (e.g. is there a path between the entrance and exit of the dungeon, or does the tree have proportions within a certain range?). If the test fails, all or some of the candidate content is discarded and regenerated, and this process continues until the content is good enough.

3 Search-based Procedural Content Generation

Search-based procedural content generation (SBPCG) is a special case of the generate-and-test approach to PCG, with the following qualifications:

– The test function does not simply accept or reject the candidate content, but grades it using one *or a vector of* real numbers. Such a test function is sometimes called a *fitness function* and the grade it assigns to the content its *fitness*.
– Generating new candidate content is contingent upon the fitness assigned to previously evaluated content instances; in this way the aim is to produce new content with higher fitness.

All of the examples below (see section 4) use some form of evolutionary algorithm (EA) as the main search mechanism. In an EA, a population of candidate content instances are held in memory. Each generation, these candidates are evaluated by the fitness function and ranked. The worst candidates are discarded and replaced with copies of the good candidates, except that the copies have been randomly modified (i.e. *mutated*) and/or recombined. However, SBPCG does not need to be married to evolutionary computation (EC); other search mechanisms are viable as well. The same considerations about representation and the search space largely apply regardless of the approach to search.

3.1 Content Representation and Search Space

A central question in EC concerns how to represent whatever is evolved. In other words, an important question is how genotypes (i.e. the data structures that are handled by the EA) are mapped to phenotypes (i.e. the data structure or process that is evaluated by the fitness function). An important distinction among representations is between *direct encodings*, wherein the size of the genotype is linearly proportional to the size of phenotype and each part of the genome maps to a specific part of the phenotype, and *indirect encodings*, wherein the genotype maps nonlinearly to the genotype and the former need not be proportional to the latter ([2–4]; see [5] for a review).

The study of representations for EC is a broad field in its own right, where several concepts have originated that bear on SBPCG [6]. The problem representation should have the right dimensionality to allow for precise searching while avoiding the "curse of dimensionality" associated with representation vectors that are too large (or the algorithm should find the right dimensionality for the vector). Another principle is that the representation should have a high *locality*, meaning that a small change to the genotype should on average result in a small change to the phenotype and a small change to the fitness value.

Apart from these concerns, of course it is important that the chosen representation is capable of representing all the interesting solutions; this ideal can be a problem in practice for indirect encodings, for which there might be areas of phenotype space to which no genotypes map.

These considerations are important for SBPCG as the representation and search space must be well-matched to the domain if it is to perform optimally. There is a continuum between SBPCG that works with direct and indirect representation. As a concrete example, a maze (for use e.g. in a “roguelike” dungeon adventure game) might be represented:

1. directly as a grid for which mutation works directly on the content (wall, free space, door, monster) of each cell,
2. more indirectly as a list of the positions, orientations and lengths of walls ([7] provides an example),
3. even more indirectly as a repository of different reusable patterns of walls and free space, and a list of how they are distributed (with various transforms such as rotation and scaling) across the grid,
4. very indirectly as a list of desirable properties (number of rooms, doors, monsters, length of paths and branching factor), or
5. most indirectly as a random number seed.

These representations yield very different search spaces. In the first case, all parts of phenotype space are reachable, as the one-to-one mapping ensures that there is always a genotype for each phenotype. Locality is likely high because each mutation can only affect a single cell (e.g. turn it from wall into free space), which in most cases changes fitness only slightly. However, because the length of the genotype would be the number of cells in the grid, mazes of any interesting size quickly encounter the curse of dimensionality.

At the other end of the spectrum, option number 5 does not suffer from search space dimensionality because it searches a one-dimensional space. However, the reason this representation is unsuitable for SBPCG is that there is no locality; one of the main features of a good random number generator is that there is no correlation between the numbers generated by different seed values. All search performs as badly (or as well) as random search.

Options 2 to 4 might all be suitable representations for searching for good mazes. In options 2 and 3 the genotype length would grow with the desired phenotype (maze) size, but sub-linearly, so that reasonably large mazes could be represented with tractably short genotypes. In option 4 genotype size is independent of phenotype size, and can be made relatively small. On the other hand, the locality of these intermediate representations depends on the care and domain knowledge with which each genotype-to-phenotype mapping is designed; both high- and low-locality mechanisms are conceivable.

3.2 Fitness Functions

Once a candidate content item is generated, it needs to be evaluated by the fitness function and assigned a scalar (or a vector of real numbers) that accurately reflects its suitability for use in the game. Designing the fitness function is ill-posed; the designer first needs to decide what, exactly, should be optimized and then how to formalize it. For example, one might intend to design a SBPCG algorithm that creates fun, immersive, frustrating or exciting game content, and thus a fitness function that reflects how much the particular piece of content contributes to the player's respective affective states while playing. At the current state of knowledge, any attempt to estimate the contribution to "fun" (or affective states that collectively contribute to player experience) of a piece of content is bound to rely on conflicting assumptions. More research is needed at this time to achieve fruitful formalisations of such subjective issues; see [8] for a review.

Three key classes of fitness functions can be distinguished for the purposes of PCG are *direct*, *simulation-based* and interactive fitness functions.

Direct Fitness Functions In a direct fitness function, some features are extracted from the generated content, and these features are mapped directly to a fitness value. Hypothetical such features might include the number of paths to the exit in a maze, firing rate of a weapon, spatial concentration of resources on an RTS map, and material balance in randomly selected legal positions for board game rule set. The mapping between features and fitness might be linear or non-linear, but typically does not involve large amounts of computation, and is typically specifically tailored to the particular game and content type. This mapping might also be contingent on a model of the playing style, preferences or affective state of the player, meaning that an element of *personalization* is possible. An important distinction within direct fitness functions is between *theory-driven* and *data-driven* functions. In theory-driven functions, the designer is guided by intuition and/or some qualitative theory of player experience to derive a mapping. On the other hand, data-driven functions are based on collecting data on the effect of various examples of content via e.g. questionnaires or physiological measurements, and then using automated means to tune the mapping from features to fitness.

Simulation-based Fitness Functions It is not always apparent how to design a meaningful direct fitness function for some game content — in some cases, it seems that the content must be sufficiently experienced and operated to be evaluated. An indirect fitness function is based on an artificial agent playing through some part of the game that involves the content being evaluated. Features are then extracted from the observed gameplay (e.g. did the agent win? How fast? How was the variation in playing styles employed?) and used to calculate the fitness of the content. The artificial agent might be completely hand-coded, or might be based on a learned behavioral model of a human player.

Another key distinction is between *static* and *dynamic* simulation-based fitness functions. In a static fitness function, it is not assumed that the agent changes while playing the game; in a dynamic fitness function the agent changes during the game and the fitness value somehow incorporates this change. For example, the implementation of the agent can be based on a learning algorithm and the fitness be dependent on *learnability*, i.e. how well and/or fast the agent learns to play the content that is being evaluated. Other uses for dynamic fitness functions is to capture e.g. order effects and user fatigue.

Interactive Fitness Functions Interactive fitness functions score content based on interaction with a player in the game, which means that fitness is evaluated during the actual gameplay. Data can be collected from the player either *explicitly*, using questionnaires or verbal input data, or *implicitly* by measuring e.g. how often or long a player chooses to interact with a particular piece of content [9], when the player quits the game, or expressions of affect such as intensity of button-presses, shaking the controller, physiological response, gaze fixation, speech quality, facial expressions and postures.

3.3 Situating Search-based PCG

At this point, let us revisit the distinctions in Section 2 and ask how they relate to SBPCG. As stated above, SBPCG algorithms are generate-and-test algorithms. They might take parameter vectors (in particular, parameters that modify the fitness function) or not. As evolutionary and similar search algorithms rely on stochasticity (e.g. a random seed is required for mutation); for the same reasons, these algorithms should be classified as stochastic rather than deterministic.

As there is no general proof that all EAs ultimately converge, there is no guaranteed completion time for a SBPCG algorithm, and no guarantee that it will produce good enough solutions. For these reasons it would seem that SBPCG would be unsuitable for online content generation, and better suited for offline exploration of new design ideas. However, as we shall see later, it is possible to successfully base complete game mechanics on SBPCG, at least if the content generated is optional rather than necessary.

We can also choose to look at the relation between indirect representation and SBPCG from a different angle. If our SBPCG algorithm includes an indirect mapping from genotype to phenotype, this mapping can be viewed as a PCG algorithm in itself, and an argument can be made for why certain types of PCG algorithms are more suitable than others for use as part of an SBPCG algorithm. It is worth noting that some indirect encodings used in various EC application areas bear strong similarities to PCG algorithms for games; several indirect encodings are based on L-systems, as are algorithms for procedural tree and plant generation [3].

4 Case Studies of Search-based PCG

In this section, we present five examples of search-based procedural content generation, and categorise those according to the distinctions made previously in the paper.

4.1 Rulesets for Pac-Man-like Games

Togelius and Schmidhuber [10] conducted an experiment in which rulesets (necessary content) were evolved offline for grid-based games in which the player moves an agent around, in a manner similar to a discrete version of Pac-Man. Apart from the agent, the grid was populated by walls and "things" of different colours, which could be interpreted as items, allies or enemies depending on the rules. Rulesets were represented fairly directly as fixed-length parameter vectors, interpreted as the effects on various things when they collided with each other or the agent, and their behaviour. A relatively wide range of games could be represented using this vocabulary, and genotype generation was deterministic except for the starting position of things. The fitness function was dynamic and simulation-based, and completely hand-crafted: an evolutionary reinforcement learning algorithm was used to learn each ruleset and the ruleset was scored dependent on how well it was learned. Games that were impossible or trivial were given low fitness, whereas those that could be learned after some time scored well.

4.2 Rulesets for Board Games

Browne [11] developed a system for offline design of rules (necessary content) for board games using a form of genetic programming. Game rules were represented relatively directly as expression trees, formulated in a custom-designed game description language. This language allowed representation of a sufficiently wide variety of board games, including many well-known games. The EA used for the creation of new rule sets was non-standard in that suboptimal children with poor performance or badly formed rules were not discarded but were instead retained in the population with a lower priority to maintain a necessary level of genetic diversity. The fitness function was a complex combination of direct measures and static simulation-based measures: for example, standard game-tree search algorithms were used to play the generated game as part of the fitness evaluation to investigate issues such as balance and time to play the game. While hand-coded, the fitness function was based on extensive study of existing board games, and measurements of user preferences for board games that exhibited various features.

4.3 Tracks for a Racing Game

Togelius et al. [12] designed a system for offline/online generation of tracks (necessary or optional content, dependent on game design) for a simple racing game. Tracks were represented directly as fixed-length parameter vectors, interpreted deterministically as b-splines (i.e. sequences of Bezier curves) that defined the course of the track. The fitness function was simulation-based, static, and personalised. Each candidate track was evaluated by letting a neural network-based car controller, which had previously been trained to drive in the style of a particular human player, drive on the track. The fitness of the track was dependent on the driving performance of the car: amount of progress, variation in progress and difference between maximum and average speed.

4.4 Weapons for a Space Shooter Game

Hastings et al. [9] developed a multi-player game built on SBPCG. In the game, players guide a spaceship through various parts of space, engaging in fire-fights with enemies and collecting weapons (each weapon is optional, but having a good set of weapons is necessary for success). Weapons are represented indirectly as variable-size vectors of real values, which are interpreted as connection topologies and weights for neural networks, which in turn control the particle systems that underlie the weapons. The fitness function is interactive, implicit and distributed. Fitness for each weapon depends on how often the various users logged on to the same server choose to fire each weapon relative to how often the weapons sit unused in users' weapon caches.

4.5 Levels and Mechanics for Super Mario Bros

Pedersen et al. [13] modified an open-source clone of the classic platform game *Super Mario Bros* to allow for personalised level and game mechanics generation. Levels were represented very indirectly as a short parameter vector describing mainly the number, size and placement of gaps in the level whereas the sole mechanic investigated was represented as the percentage of the level played from right to left. This vector was converted to a complete level in a stochastic fashion. The fitness function was direct, data-driven and personalised, using a neural network that converted level parameters and information about the player's playing style to one of six emotional state predictors (fun, challenge, frustration, predictability, anxiety, boredom), which could be chosen as components of a fitness function. These neural networks were trained through collecting both gameplay metrics and data on player preferences using variants of the game on a web page with an associated questionnaire.

5 Outlook

As reviewed in the previous section, a small number of successful experiments are already beginning to show the promise of search-based procedural content generation. By classifying these experiments according to the taxonomies presented in this paper, it can be seen both that (1) though all are examples of SBPCG, they differ from each other in several important dimensions, and (2) there is room for approaches other than those that have already been tried; both the type of content generated and the algorithmic approach to generating it may change in the future.

At the same time, there are several hard and interesting research challenges. These include the appropriate representation of game content and the design of relevant, reliable, and computationally efficient fitness functions. The latter challenge in particular is likely to benefit from collaboration with experts from fields other than computational intelligence, including psychology, game design studies and affective computing. The potential gains from providing good solutions to these challenges, however, are significant: the invention of new game genres built on PCG, streamlining of the game development process, and further understanding of the mechanisms of human entertainment are all possible.

Acknowledgements

Thanks to all the participants in the discussions in the Procedural Content Generation Google Group. The research was supported in part by the Danish Research Agency, Ministry of Science, Technology and Innovation; project name: *AGameComIn*; project number: 274-09-0083.

References

1. Miller, G.S.P.: The definition and rendering of terrain maps. In: Proceedings of SIGGRAPH. Volume 20. (1986)
2. Bentley, P.J., Kumar, S.: The ways to grow designs: A comparison of embryogenies for an evolutionary design problem. In: Proceedings of the Genetic and Evolutionary Computation Conference. (1999) 35–43
3. Hornby, G.S., Pollack, J.B.: The advantages of generative grammatical encodings for physical design. In: Proceedings of IEEE CEC. (2001)
4. Stanley, K.O.: Compositional pattern producing networks: A novel abstraction of development. Genetic Programming and Evolvable Machines Special Issue on Developmental Systems **8**(2) (2007) 131–162
5. Stanley, K.O., Miikkulainen, R.: A taxonomy for artificial embryogeny. Artificial Life **9**(2) (2003) 93–130
6. Rothlauf, F.: Representations for Genetic and Evolutionary Algorithms. Springer, Heidelberg (2006)
7. Ashlock, D., Manikas, T., Ashenayi, K.: Evolving a diverse collection of robot path planning problems. In: Proceedings of IEEE CEC. (2006) 6728–6735
8. Yannakakis, G.N.: How to Model and Augment Player Satisfaction: A Review. In: Proceedings of the 1st Workshop on Child, Computer and Interaction, Chania, Crete, ACM Press (2008)
9. Hastings, E., Guha, R., Stanley, K.O.: Evolving content in the galactic arms race video game. In: Proceedings of the IEEE Symposium on Computational Intelligence and Games. (2009)
10. Togelius, J., Schmidhuber, J.: An Experiment in Automatic Game Design. In: Proceedings of the IEEE Symposium on Computational Intelligence and Games, Perth, Australia, IEEE (2008) 252–259
11. Browne, C.: Automatic generation and evaluation of recombination games. PhD thesis, Queensland University of Technology (2008)
12. Togelius, J., De Nardi, R., Lucas, S.M.: Towards automatic personalised content creation in racing games. In: Proceedings of the IEEE Symposium on Computational Intelligence and Games. (2007)
13. Pedersen, C., Togelius, J., Yannakakis, G.N.: Modeling Player Experience in Super Mario Bros. In: Proceedings of the IEEE Symposium on Computational Intelligence and Games, Milan, Italy, IEEE (2009) 132–139