# Chapter 1
# The search-based approach (DRAFT)

Julian Togelius and Noor Shaker

## 1.1 What is the search-based approach to procedural content generation?

There are many different approaches to generating content for games. In this chapter, we will introduce the *search-based approach*, which has been intensely investigated in academic PCG research in recent years. In search-based procedural content generation, an evolutionary algorithm or some other stochastic search/optimisation algorithm is used to search for content with the desired qualities in a search space. The basic metaphor is that of design as a search process: a good enough solution to the design problem exists within some space of solutions, and if we keep iterating and tweaking on one or many possible solutions, keeping those changes which make the solution(s) better and discarding those that are harmful, we will eventually arrive at the desired solution. This metaphor has been used to describe the design process in many different disciplines: for example, Will Wright (designer of *SimCity* and *The Sims*) described the game design process as search in his talk at Game Developers Conference 2004.

The core components of a search-based approach to solving a content generation problem are the following:

- A *search algorithm*. This is the "engine" of a search-based method. As we will see, often relatively simple evolutionary algorithms work well enough, though sometimes there are substantial benefits to using more sophisticated algorithms that take e.g. constraints into account, or that are specialised for a particular content representation.
- A *content representation*. This is the representation of the artefacts you want to generate, e.g. levels, quests or winged kittens. The content representation could be anything from an array of real numbers to a graph to a string. The content representation defines (and thus also limits) what content can be generated, and determines whether effective search is possible.

- One or more *evaluation functions*. An evaluation function is a function from an artefact (an individual piece of content) to a number indicating the quality of the artefact. The output of an evaluation function could indicate e.g. the playability of a level, the intricacy of a quest or the aesthetic appeal of a winged kitten. Crafting an evaluation function that reliably measures the aspect of game quality that it is meant to measure is often among the hardest tasks in developing a search-based PCG method.

This chapter will describe each of these components in turn. It will also discuss several examples of search-based methods for generating different types of content for different types of games.

## 1.2 Evolutionary search algorithms

An evolutionary algorithm is a stochastic search algorithm loosely inspired by Darwinian evolution through natural selection. The core idea is to keep a *population* of *individuals* (also called chromosomes or candidate solutions), which in each *generation* are evaluated, and the *fittest* (highest evaluated) individuals are allowed to *reproduce* and the least fit are removed from the population. A generation can thus be seen as divided into *selection* and reproduction phases. In your backyard, a generation of newly born rabbits may be subject to selection by the hungry wolf who eats the slowest of the litter, with the surviving rabbits being allowed to reproduce. The next generation of rabbits is likely to, on average, be better at running from the wolf. Similarly, in your search-based PCG implementation, a generation of strategy game units might be subject to selection by an evaluation function that grades them based on how complementary they are, and then mixed with each other (*recombination* or *crossover*) or copied with small random changes (*mutation*). The next generation of strategy game units is likely to, on average, be more complementary. It is important to note that this process works even when the initial generation consists of randomly generated individuals which are all very unfit for the purpose; some individuals will be less worthless than others, and a well-designed evaluation function will reflect these differences.

To make matters more concrete, let us describe a simple but fully usable evolutionary algorithm, the $\mu + \lambda$ *evolution strategy* (ES). The parameter $\mu$ represents the size of the part of the population that is kept between generations, the *elite*; the parameter $\lambda$ represents the size of the part of the population that is generated through reproduction each generation. For simplicity, imagine that $\mu = \lambda = 50$ while reading the following description.

1. Initialise the population of $\mu + \lambda$ individuals. The individuals could be randomly generated, or include some individuals that were hand-designed or the result of previous evolutionary runs.
2. Shuffle the population (permute it randomly). This phase is optional but helps escaping loss-of-gradient situations.

3. Evaluate all individuals with the evaluation function, or some combination of several evaluation functions, so that each individual is assigned a single numeric value indicating its fitness.
4. Sort the population in order of ascending fitness.
5. Remove the $\lambda$ worst individuals.
6. Replace the $\lambda$ removed individuals with copies of the $\mu$ remaining individuals. The newly made copies are called the *offspring*. If $\mu = \lambda$, each individual in the elite is copied once; otherwise, it could be copied fewer or more times.
7. Mutate the $\lambda$ offspring, i.e. perturb them randomly. The most suitable mutation operator depends on the representation and to some extent on the fitness landscape. If the representation is a vector of real numbers, an effective mutation operator is *Gaussian mutation*: add random numbers drawn from a Gaussian distribution with a small standard deviation to all numbers in the vector.
8. If the population contains an individual of sufficient quality, or the maximum numbers of generations is reached, stop. Otherwise, go to step 2 (i.e. start the next generation).

Despite the simplicity of this algorithm (it could be implemented in 10-20 lines of code), the $\mu + \lambda$ ES can be remarkably effective; even degenerate versions such as the $1 + 1$ ES can work well. However, the evolution strategy is just one of several types of evolutionary algorithms; another commonly used type is the genetic algorithm, which relies more on recombination and less on mutation, and which uses different selection mechanisms. There are also several types of stochastic search/optimisation algorithms that are not strictly speaking evolutionary algorithms but can be used for the same purpose, e.g. swarm intelligence algorithms such as particle swarm optimisation and ant colony optimisation. A good overview of evolutionary algorithms and some related approaches can be found in Eiben and Smith's book [7].

Some evolutionary algorithms are especially well suited to particular types of representation. For example, numerous variations on evolutionary algorithms have been developed especially for evolving runnable computer programs, often represented as expression trees [17]. If the artefacts are represented as vectors of real numbers of relatively short length (low dimensionality), a particularly effective algorithm is the Covariance Matrix Adaptation Evolution Strategy (CMA-ES), for which there are several open source implementations available [8].

In many cases we want to use more than one evaluation function, as it is hard to capture all aspects of an artefact's quality in one number. For using a standard single-objective evolutionary algorithm such as the evolution strategy, the evaluation functions could be combined as a weighted sum. However, this comes with its own set of problems, particularly that some functions tend to be optimised at the expense of others. Instead, one could use a *multiobjective* evolutionary algorithm, that optimises for several objectives at the same time and finds the set of *nondominated* individuals which have unique combinations of strengths. The perhaps most popular multiobjective evolutionary algorithm is the NSGA-II [6].

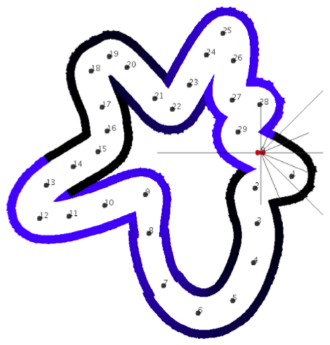

Fig. 1.1: A track evolved based on sequences of Bezier curves. Taken from [23]

## 1.3 Content representation

Content representation is a vitally important issue when evolving game content. The representation chosen plays an important role in the efficiency of the generation algorithm and the space of content the method will be able to cover. In evolutionary algorithms, the solutions in the generation space are usually encoded as *genotypes* which are used for efficient searching and evaluation. Genotypes are later converted to *phenotypes*; the actual entities being evolved. In a game content generation scenario, the genotype might be the instructions for creating a game level, and the phenotype is the actual game level.

Examples about content representation in the game domain include the work done by Togelius et. al. [25] who exploited two forms of representation for evolving maps for *StarCraft* [1]: an indirect representation was used for searching (an array of different map elements such as bases, resources and impassable areas) and a direct representation (a heightmap grid) for quality testing and visualisation.

In another game genre, Cardamone et. al. [3] conducted a study for evolving tracks for a car racing game using a set of control points the track has to cover and Bezier curves were employed to connect these points and ensure smoothness, a method inspired by the work done by Togelius et. al. [23] on the same game genre. An example track evolved following this method is presented in Figure 1.1.

As a concrete example of different representations, a level in Super Mario Bros might be represented:

1. directly as a level map, where each variable in the genotype corresponds to one “block” in the phenotype (e.g. bricks, question mark blocks, etc.),
2. more indirectly as a list of the positions and properties of the different game entities such as enemies, platforms, gaps and hills (an example of this can be found in [18]),
3. even more indirectly as a repository of different reusable patterns (such as collections of coins or hills), and a list of how they are distributed (with various

transforms such as rotation and scaling) across the level map (an example of this can be found in [22]),

4. very indirectly as a list of desirable properties such as number of gaps, enemies, coins, width of gaps (an example of this can be found in [19]), or
5. most indirectly as a random number seed.

These representations yield very different search spaces. It's easy to think that the best representation would be the most direct one, which gives the evolutionary process most control over the phenotype. One should be aware, however, of the "curse of dimensionally" associated with representations that yield large search spaces: the larger the search space, the harder it might be to find a certain solution. Another useful 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. In that sense, the last representation is unsuitable for search-based PCG because there is no locality and in this case, all search methods perform as badly (or as well) as random search.

The choice of proper representation depends on the type of problem one is trying to solve. In the work done by Shaker et. al. [19], the levels of *Infinite Mario Bros* [16], a public clone of the popular game *Super Mario Bros* [13], are represented according to option 4 as a vector of integers; each level is parametrized by four selected content features with the intension of finding the best combination of these features that can be used to generate content that optimises specific experience for a particular player. In a latter study by the same authors [18], a more expressive representation is used following option 2, in which the structure of the levels of the same game was described in a Design Grammar, written in Backus-Naur Form which specifies the type, position and properties of each item to be placed in the level map. The design grammar is latter employed by Grammatical Evolution [14] to evolve level design. A set of design elements, following option 3, was proposed in [22], also on the same game, where levels were described as a list of design elements placed in 2D maps, and in this study standard genetic algorithm was used to evolve content.

An issue closely related to the representation on the direct-indirect continuum is the expressive range of the chosen representation. The expressive range is relative to a particular measure of it: one could measure the expressivity of a platform game level generator in terms of how many different configurations of blocks it could produce, but it would make more sense to measure some quality that is more relevant to the experience of playing the game as a human. For example, the four-feature vector representation used to represent Infinite Mario Bros levels allows control of the generation over only the four dimensions chosen, and consequently the search space is bounded by the range of these four features. On the other hand, a generator with a wider expressive range was built when representing the possible level designs in a design grammar which imposes less constrains on the structures evolved.

## 1.4 Evaluation functions

Once a proper representation is found and candidate solutions are generated, it needs to be evaluated by an evaluation function to assign a score (a fitness) for each candidate based on which the process of evolution will continue. In general, the evaluation function should be designed to model some desirable quality of the artefact, e.g. it's playability, regularity, entertainment value etc. The design of a fitness function depends to a great extend on the designer and what she think are the important aspects that should be optimised and how to formulate that.

For example, there exists many studies on evolving game content that is "fun" [24, 23, 19, 3]. This term, however, is not well defined and hard to measure and formalise. This problem has been approached by many authors from different perspectives. In some studies, fun is considered a function of player behaviour and it is measured accordingly. An example of such method can be found in the work done by Togelius et. al. [24] for evolving entertaining car racing tracks. In this study, indicators of player performance, such as the average speed achieved, were used as a measure of suitability of each evolved track for individual players. In another study by Shaker et. al. [19], fun is measured through self reports by directly asking the players about their experience. In other studies [21], a game is considered fun if the content presented follows predefined patterns that specify regions in the game and alternate between segments of varying challenge. In this case, challenge is considered the primary cause of a fun experience.

In search-based PCG, one commonly distinguishes between three classes of evaluation functions:

1. Direct: Direct evaluation functions map features extracted from the content generated to a content quality value and in that sense, they base their fitness calculations directly on the phenotype representation of the content. Direct evaluation functions are fast to compute and often relatively easy to implement, but it is sometimes hard to devise a direct evaluation function for some aspects of game content. Example features include the placement of bases and resources in real-time strategy games [25] or the size of the ruleset in strategy games [11]. The mapping between features and fitness might be contingent on a model of the playing style, preferences or affective state of players. An example of this form of fitness is the study done by Shaker et. al. [19, 20] for personalising player experience using models of players as a measures of content quality.
   Within direct evaluation functions, one distinguishes between *theory-driven* and *data-driven* functions. Theory-driven functions are guided by intuition and/or qualitative theories of player experience. Togelius et. al. [23] used this method to evaluate the tracks in a car racing game. The fitness function derived is based on several theoretical studies on fun in games [5, 10] and the authors' intuition of what makes an intertraining track. Data-driven functions, on the other hand, are based on quantitative measures of player experience that approximate the mapping between the content presented and players' affective or cognitive states collected via questionnaires or physiological measurements [20, 28].

2. Simulation-based: Simulation-based evaluation functions utilise AI agents that play through the content generated and estimate its quality. Statistics are usually calculated about the agents behaviour and playing style and used to score game content. The type of the evaluation task determines the area of proficiency of the AI agent; if content is evaluated on the basis of playability, that is the existence of a path from the start to the end in a maze or a level in a 2D platform game, then AI agents should be designed that are excel in reaching the end of the game. On the other hand, if content is optimised to maximise particular player experience, then an AI agent that imitates human behaviour is usually adopted. An example study that implement human-like agent for assessing content quality is presented in [23] where neural network-based controllers are trained to drive like human players in a car racing game and then used to evaluate the generated tracks. Each track generated is given a fitness value according to playing behaviour statistics calculated while the AI controller is playing. Another example of a simulation-based evaluation function is measuring the average fighting time of bots in a first-person shooter game [4].
   An important distinction within simulation-based fitness functions is between *static* and *dynamic* functions. Static evaluation functions assume that the agent behaviour is maintained during gameplay. This assumption is not held in dynamic evaluation functions which incorporate the changes in the agent's behaviour during gameplay. In such agents, the fitness value can be dependent on learnability: how well and/or fast the agent learns to play the content that is being evaluated.
3. Interactive: Interactive functions evaluate content based on interaction with a human, so they requires a human "in the loop". Examples of this method can be found in the work done by Hasting et. al. [9] who implemented this approach by evaluating the quality of the personalised weapons evolved implicitly based on how often and how long the player chooses to use these weapons. Cardamone et. al. [3] also used this form of evaluation to score racing tracks according to the users' reported preferences. The first case is an example of an *implicit* collection of data while players' preferences were collected *explicitly* in the second. The problem with explicit data collection is that it usually requires interrupting the gameplay session if not well integrated. This method however provides a reliable and accurate estimator of player experience as opposite to implicit data collection which is usually noisy and based on assumptions. Hybrid approach are sometimes employed to accommodate for the drawbacks of these two methods by collecting information across multiple modalities such as combining player behaviour with eye gaze and/or skin conductance. Example studies that use this approach can be found in [12, 20, 28].

## 1.5 Example: StarCraft maps

In two recent papers, Togelius et al. presented a search-based approach to generating maps for the classic real-time strategy game (RTS) *StarCraft* [25, 26]. Despite being released in the previous millennium, this game is still widely played and was until very recently the focus of large tournaments broadcast on national TV in countries such as South Korea. The focus of the game is on building bases, collecting resources, and waging war with armies of units built in the bases. The maps of the game play a crucial role, as they constrain what strategies are possible through their distribution of paths, obstacles, resources etc. Given the competitive nature of the game, it is very important that the maps are fair. Therefore, evaluation functions were designed to measure the fairness of the maps as well as their affordances for interesting and diverse strategies.

**Representation:** The maps are represented as vectors of real numbers (of around 100 dimensions). In the genotype-to-phenotype process, some of these numbers are interpreted directly as the positions of resources or base starting locations. Other numbers are interpreted as starting positions and parameters for a turtle-graphics-like procedure that "draws" impassable regions (walls, rocks, etc.) on the initially empty map. The result of the transformation is a two-dimensional array where each cell corresponds to a block in the StarCraft map format; this can then automatically be converted to a valid StarCraft map.

**Evaluation:** Eight different evaluation functions were developed that address base placement, resource placement and paths between bases. These evaluation functions are based mostly on calculations of free space in different areas of the map and on the shortest paths between different points as calculated by the A* algorithm, and the functions are thus direct (though, if you see the path calculations as abstract simulations of unit behaviour in the game, the functions can be seen as simulation-based). There are functions for evaluating that bases are sufficiently fair from each other, that there is enough space to grow a base, and that there is equal access to nearby resources. One particularly complicated function is the choke point function, which returns a higher value if the shortest path between two bases has a so called choke point, where a tactically skilled player can defend against a superior attacking forces through using the level geometry.

**Algorithm:** Given the number of evaluation functions, it seemed very complicated to combine all of them into a single objective. The SMS-EMOA, a state of the art multiobjective evolutionary algorithms, was therefore used to evolve combinations of two or three objectives (some additional objectives were also converted to constraints). It was found that there are partial conflicts between several objectives, meaning that it is impossible to find a map that maximises all of them, but certain combinations of objectives yield interesting and reasonably fair maps.

## 1.6 Example: Racing tracks

In an early and influential paper, Togelius et al. evolved racing tracks to fit particular players' playing style in a simple two-dimensional racing game [23]. This particular game had already been used for a series of experiments investigating how evolutionary algorithms could best be used to create neural networks that could play the game well, when the authors decided to see whether the same technique could be applied to evolve the tracks the car was racing on. The reasoning was that creating challenging opponent drivers for commercial racing games is actually quite easy, especially if you are allowed to “cheat” by giving the computer-controlled cars superior performance (and who would stop you?) – on the other hand, creating an interesting racing track is not trivial at all.

**Representation:** The tracks are represented as vectors of real numbers, which are interpreted as control points for b-splines, i.e. sequences of Bezier curves.

**Evaluation:** The tracks are meant to be personalised for individual players. Therefore, the first stage in evolving a track for a given player is to model the playing style of that player. This is done by teaching a neural network (via another evolutionary process) to drive like that player. Then a candidate track is evaluated in a simulation-based manner by letting the neural network driver drive on that track in lieu of the human player and investigate its performance. This information is used by three different evaluation functions, that measure whether the track has appropriate challenge and diversity for the player.

**Algorithm:** Given that there are three different evaluation functions, there remains the problem of combining them. The algorithm used, *cascading elitism*, is similar to $\mu + \lambda$ ES but has several stages of selection to ensure appropriate selection pressure on all objectives.

## 1.7 Example: Board game rules

In an influential paper, Browne and Maire demonstrated that it is possible to automatically generate complete board games of such quality that they can be sold as commercial products [2]. The system described, *Ludi*, is restricted to simple board games similar to Go, Othello and Connect Four, but does a remarkable job of exploring this search space. This example will be discussed further in chapter **??**.

**Representation:** The board games, including board layouts and rules, are represented as strings (which can be interpreted as expression trees) in a special purpose game description language. This is a relatively high-level language, describing entire games in just a few lines.

**Evaluation:** The games were evaluated by playing them with a version of the MiniMax algorithm, with an evaluation function that had been automatically tuned for each game. A number of values are extracted from investigating the performance of the algorithm on the game, e.g. how long time it took to finish the game, how often the game ends in a draw, how many of the rules were used etc. These values were combined using a weighted sum based on empirical investigations of the properties of successful board games.

**Algorithm:** A relatively standard genetic algorithm was used.

## 1.8 Example: Galactic Arms Race

Galactic Arms Race (GAR) is a space shooter video game where the player traverses the space in a space ship, shoots enemies, collects items and upgrades her ship. The game was first released in 2010 as a free research game and a commercial version of the game was recently released in 2012. The game is interesting from a research perspective because it is one of very few games, if any, that incorporate online automatic personalised content generation in a highly and very well-chosen playable context. The main innovation of the game is in personalising the weapons used by the player through evolution. As the game is played, new particle weapons are automatically generated based on player behaviour.

**Representation:** Particle system weapons are controlled by Artificial Neural Networks (ANNs) evolved by a method called NeuroEvolution of Augmenting Topologies (NEAT) [9] which evolves neural network through complexification; a term referring to starting the evolution with a population of simple, small networks and increasing the complexity of the network topologies over generations. Each weapon in the game is represented as a single ANN that control the motion (velocity) and appearance (colour) of the particles given the particle current position in the space. The evolution starts with a set of simple weapons that shoot only in a straight line.

**Evaluation:** During the game, a fitness is assigned for each weapon based on how much this particular weapon is used by the player. Not picking up a weapon causes its ineligibility for reproduction. The weapons used by the user are assigned higher fitness values and thus have higher probability of being evolved. The newly evolved content are then spawned in the space for the player to pick up.

**Algorithm:** The whole game thus represents a collective, distributed evolutionary algorithm. This process allows the generation of unique weapons for each player and by playing the game, more novel and personalised weapons based on those preferred could be found.

## 1.9 Lab exercise: Evolve a dungeon

Roguelike games are a type of games that use PCG for level generation. Following the original game Rogue from 1980, a roguelike typically lets you control an agent in a labyrinthine dungeon, collecting treasures, fighting monsters and levelling up. A level in such game thus consists of rooms of different sizes containing monsters and items and connected by corridors. There are a number of standard constructive algorithms for generating roguelike dungeons, such as [15]:

- Create the rooms first and then connect them by corridors or,
- use maze generation methods to create the corridors and then connect adjacent sections to create rooms.

The purpose of this exercise is to allow you to understand the search-based approach through implementing a search-based dungeon generator. Your generator should evolve playable dungeons for an imaginary roguelike. The phenotype of the dungeons should be 2D matrices (e.g. size $50x50$) where each cell could be one of the following: free space, wall, starting point, exit, monster, treasure. It is up to you whether to add other possible types of cell content, such as traps, teleporters, doors, keys, or different types of treasures and monsters. One of your tasks is to explore different content representations and quality measures in the context of dungeon generation. Possible content representations include [27]:

- A grid of cells that can contain one of the different items including: walls, items, monsters, free spaces and doors,
- a list of walls with their properties including their position, length and orientation,
- a list of different reusable patterns of walls and free space, and a list of how they are distributed across the grid,
- a list of desirable properties (number of rooms, doors, monsters, length of paths and branching factor), or
- a random number seed.

There are a number of advantages and disadvantages for each of these representations. In the first representation, for example, a grid of size $100 \times 100$ would need to be encoded as a vector of length 10,000, which is more than many search algorithms can effectively approach. The last option, on the other hand, explores one-dimensional space but it has no locality.

Content quality can be measured directly by counting the number of unreachable rooms or undesired properties such as a corridor connected to a corner in a room or a room connected to too many corridors.

## References

1. Blizzard Entertainment, Mass Media: (1998). StarCraft, Blizzard Entertainment and Nintendo
2. Browne, C., Maire, F.: Evolutionary game design. IEEE Transactions on Computational Intelligence and AI in Games, (1), 1–16 (2010)
3. Cardamone, L., Loiacono, D., Lanzi, P.L.: Interactive evolution for the procedural generation of tracks in a high-end racing game. In: Proceedings of the 13th annual conference on Genetic and evolutionary computation, pp. 395–402. ACM (2011)
4. Cardamone, L., Yannakakis, G.N., Togelius, J., Lanzi, P.: Evolving interesting maps for a first person shooter. Applications of Evolutionary Computation pp. 63–72 (2011)
5. Csikszentmihalyi, M.: Flow: The Psychology of Optimal Experience. Harper Perennial (1991)
6. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: Nsga-ii. Evolutionary Computation, IEEE Transactions on **6**(2), 182–197 (2002)
7. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer (2003)
8. Hansen, N., Ostermeier, A.: Completely derandomized self-adaptation in evolution strategies. Evolutionary computation **9**(2), 159–195 (2001)

9. Hastings, E.J., Guha, R., Stanley, K.: Evolving content in the galactic arms race video game. In: Proceedings of the 5th international conference on Computational Intelligence and Games, pp. 241–248. IEEE (2009)
10. Koster, R.: A theory of fun for game design. Paraglyph press (2004)
11. Mahlmann, T., Togelius, J., Yannakakis, G.N.: Modelling and evaluation of complex scenarios with the strategy game description language. In: Computational Intelligence and Games (CIG), 2011 IEEE Conference on, pp. 174–181. IEEE (2011)
12. Martinez, H., Yannakakis, G.N.: Mining multimodal sequential patterns: A case study on affect detection. In: Proceedings of the 13th International Conference in Multimodal Interaction. ACM (2011)
13. Nintendo Creative Department: (1985). Super Mario Bros, Nintendo
14. O'Neill, M., Ryan, C.: Grammatical evolution. IEEE Transactions on Evolutionary Computation (4), 349–358 (2001)
15. PCG Wiki: Procedural content generation wiki. URL http://pcg.wikidot.com/
16. Persson, M.: Infinite mario bros. URL http://www.mojang.com/notch/mario/
17. Poli, R., Langdon, W.W.B., McPhee, N.F., Koza, J.R.: A field guide to genetic programming. Lulu. com (2008)
18. Shaker, N., Nicolau, M., Yannakakis, G.N., Togelius, J., ONeill, M.: Evolving levels for super mario bros using grammatical evolution. pp. 304–311 (2012)
19. Shaker, N., Togelius, J., Yannakakis, G.N.: Towards automatic personalized content generation for platform games. In: Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE) (2010)
20. Shaker, N., Yannakakis, G.N., Togelius, J., Nicolau, M., ONeill, M.: Fusing visual and behavioral cues for modeling user experience in games. IEEE Transactions on System Man and Cybernetics; Part B: Special Issue on Modern Control for Computer Games
21. Sorenson, N., Pasquier, P.: The evolution of fun: Automatic level design through challenge modeling. In: Proceedings of the First International Conference on Computational Creativity (ICCCX). Lisbon, Portugal: ACM, pp. 258–267 (2010)
22. Sorenson, N., Pasquier, P., DiPaola, S.: A generic approach to challenge modeling for the procedural creation of video game levels. IEEE Transactions on Computational Intelligence and AI in Games (3), 229–244 (2011)
23. Togelius, J., De Nardi, R., Lucas, S.: Towards automatic personalised content creation for racing games. In: IEEE Symposium on Computational Intelligence and Games, 2007. CIG 2007, pp. 252–259. IEEE (2007)
24. Togelius, J., Nardi, R.D., Lucas, S.M.: Making racing fun through player modeling and track evolution. In: Proceedings of the SAB'06 Workshop on Adaptive Approaches for Optimizing Player Satisfaction in Computer and Physical Games (2006)
25. Togelius, J., Preuss, M., Beume, N., Wessing, S., Hagelbäck, J., Yannakakis, G.N.: Multiobjective exploration of the starcraft map space. In: Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG), pp. 265–272. Citeseer (2010)
26. Togelius, J., Preuss, M., Beume, N., Wessing, S., Hagelbäck, J., Yannakakis, G.N., Grappiolo, C.: Controllable procedural map generation via multiobjective evolution. Genetic Programming and Evolvable Machines pp. 1–33 (2013)
27. Togelius, J., Yannakakis, G.N., Stanley, K.O., Browne, C.: Search-based procedural content generation. In: Proceedings of EvoApplications. Springer LNCS (2010)
28. Yannakakis, G.N., Hallam, J.: Entertainment modeling in physical play through physiology beyond heart-rate. Affective Computing and Intelligent Interaction pp. 254–265 (2007)