Search-Based Procedural Content Generation in Games

Ian L. McGathey
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
mcgat003@morris.umn.edu

ABSTRACT

This paper discusses search-based procedural content generation (SBPCG) systems used to generate content in videogames. Procedural content generation (PCG) involves generating content algorithmically, using some form of randomization or psuedo-randomization to create a variety of different content pieces. Search-based procedural content generation involves using a fitness function to test and score individual content pieces after they are generated. The example of in this paper discusses implementing SBPCG through the use of a genetic algorithm (GA). An applications section is also included, which discusses the implementation and testing of two SBPCG systems: a system for generating racetracks in The Open Racing Car Simulator (TORCS) proposed by Cardamone *et al.*, and a system proposed by Mourato *et al.* for generating levels in *Prince of Persia*.

Keywords

genetic programming, procedural content generation, randomly generated content, videogame development

1. INTRODUCTION

Modern videogame developers are constantly pressured to deliver a product that is desirable to consumers. Gamers expect videogames to include elements such as a large game world to explore, access to many different unique items and weapons, or a game world that is different each time the game is played. Since implementing these elements can be a daunting task for developers, a technique called procedural content generation (PCG) is often used to automate much of the work. Procedural content generation relies on algorithms that incorporate some level of randomness to generate tangible pieces of videogame content [5]. A common issue with basic PCG is that the content produced may be of lower quality than content specifically designed by a developer. A procedurally generated track in a car racing game, for example, may not be nearly as fun to drive on as one designed by a developer. Search-based procedural content generation (SBPCG) aims to solve this problem by testing pieces of generated content and scoring them based on criteria specified by developers. This is primarily done by generating content within a genetic algorithm. This way, future generations of generated content are more likely to meet the expectations of the players and developers.

UMM CSci Senior Seminar Conference, December 2014 Morris, MN.

2. BACKGROUND

Before discussing search-based procedural content systems in videogames, it is important to first understand procedural content generation and some related concepts.

2.1 Procedural Content Generation

Procedural content generation (PCG) is process in which an algorithm is used to automate content creation [5]. These algorithms are referred to as content generation algorithms. For this process to work, content generation algorithms are designed so they introduce some level of randomness in generated content. For example, if an algorithm is used to generate separate areas within a game work, it may randomly generate coordinates that correspond to the placement of certain features like buildings, trees, mountains etc. This randomness allows for variety in the content created. Individual items produced by a PCG system will be referred to as "content pieces".

2.2 Content Representation

In order for content to be generated, it must be represented in a specific way. We will discuss a basic example if a PCG system that uses templates and experimental chunks to build content pieces.

2.2.1 Templates

Templates represent the general structure of content pieces, but lack specific parts [4]. They typically contain placeholders that are later replaced by experimental chunks to form content pieces. An example if a template would be an empty grid of placeholder tiles that form the general structure of a two-dimensional game map.

2.2.2 Experimental Chunks

Experimental chunks are items that are used to fill in templates to create content pieces [4]. They are often human designed, but can also be procedurally generated. An example of experimental chunks would be pieces of geometry (floors, walls, etc.) that are used to fill in the placeholder tiles within the template example described in section 2.2.1.

2.2.3 Online vs. Offline PCG

Two large categories of PCG are online and offline [5]. Online refers to content being generated as the player is interacting with the game world. An example would be an algorithm that, as the player moves though the game world, spawns enemies just out of the players sight. Offline refers to PCG that occurs before the player interacts with the game. This can mean that such content is generated while the game or a particular map or level is loading. An example of this could be a game where the environments are randomly generated every time the game loads, giving the player a different experience each time they play. It also includes PCG that is done by developers when building the game. For example, a game may include a very large and complex game world that is always the same to players, but was initially developed using PCG.

2.2.4 Constructive vs. Generate-and-test PCG

Two more categories that procedural content generation algorithms are broken down into are constructive PCG and generate-and-test PCG [5]. In constructive PCG system, the content generation algorithm outputs finished content pieces. No testing is done after the content pieces are generated. In generate-and-test PCG, this is not the case. Content pieces that are generated by the content generation algorithm are run through a test, and either accepted or rejected. Accepted content pieces are used, and rejected ones are not. See Figure 1.

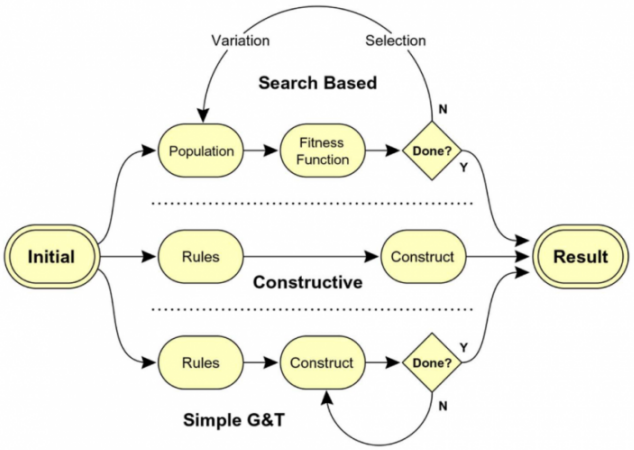

Figure 1: This flow diagram displays a comparison between search-based, constructive, and generate-and-test PCG [6].

3. SEARCH-BASED PCG

Search-based procedural content generation (SBPCG) is a special type of generate-and-test PCG. It differs from the standard generate-and-test approach in that content pieces are not simply accepted or rejected during the testing phase. Instead, these pieces of content are tested using a fitness function, and assigned fitness values that correspond to their suitability [5]. See Figure 1. The following sections will discuss how an SBPCG system can be created by using a genetic algorithm.

4. GENETIC ALGORITHMS

Geneitic algorithms (GA) are search algorithms that mimic the processes of evolution and natural selection in order to find a solution to a problem [2]. Search-based PCG systems are most commonly implemented through the use of genetic algorithms. The following subsections detail some primary concepts and components of a genetic algorithm used in an SBPCG system.

4.1 Fitness Functions

A fitness functions tests the individuals within a population, and assigns them fitness values [5]. These fitness values correspond to the suitability of the individuals based on criteria specified by developers within the fitness function. When creating fitness functions, developers aim to include criteria that best represents an ideal solution to the problem. In the context of search-based PCG, fitness functions aim to help find the best suited content pieces.

4.1.1 Direct

Direct fitness functions are the most simple, and usually require little computation. In a direct fitness function, certain attributes of an individual are extracted and examined [5]. A fitness value is then assigned based on these attributes. Assigned fitness values may correspond to specific, individual characteristics, or to combinations of characteristics. The relation between characteristics and fitness values may also be linear or non-linear. A simple example of a direct fitness could be one that scores randomly generated NPC characters in a medieval-themed role playing game with the attributes (strength, speed, agility, intelligence, and dexterity). If the goal is to find the best warrior-type character, the fitness value may be an individuals strength value, plus their dexterity value.

4.1.2 Simulation-Based

As the name implies, simulation-based fitness tests score content based on the performance of that content within a simulation [5]. This is often done because it is not always apparent how well content will be suited to certain situation, and therefore it is difficult to design a direct fitness function to test it. Simulation-based fitness functions are especially useful when testing artificial intelligence, but can also be used for other applications as well.

4.1.3 Interactive

Interactive fitness functions score content based on the player's interactions or input [5]. This can be done by collecting data while the player is playing the game, or by explicitly asking the player using a some sort of form or questionnaire about their experiences.

4.2 Genetic Operators

Genetic operators are functions within a genetic algorithm that are used in order to introduce variation to the population during subsequent generations [7].

4.2.1 Selection

Selection operators are modeled after the theory of natural selection, and determine which individuals will be selected to add variation to the population. Two common selection operators are truncation and tournament selection [8]:

1. **Truncation Selection.** In truncation selection, individuals in the population are ordered based on their fitness values. A proportion of the population with the highest fitness values is selected for crossover.
2. **Tournament Selection.** Tournament selection involves running "tournaments" where randomly chosen individuals from the population are picked. The individual with the highest fitness value in each tournament "wins", and is selected for crossover.

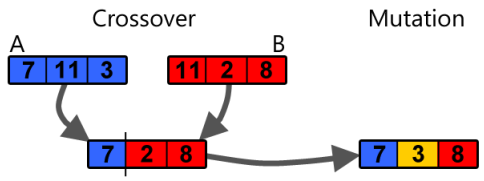

Figure 2: Examples of crossover and mutation represented using integer arrays of size 3. Crossover is performed on individuals A and B to create a new individual. Mutation is then performed on that new individual [3].

4.2.2 *Crossover and Mutation*

The crossover and mutation operators model different aspects of biological reproduction [7]:

1. **Crossover.** This operator mimics the creation of a new individual through biological reproduction. Two "parent" individuals in the population are used to create a new individual. When crossover occurs, a new individual is introduced into the population that takes part of its characteristics from one parent individual, and part from the other. See Figure 2.
2. **Mutation.** The mutation operator is modeled after biological mutation of a new individual. It is performed on a single individual, typically one that has been created as a result of crossover. One of the traits of the newly created individual is replaced by another randomly selected trait. See Figure 2.

4.3 Genetic Algorithms and SBPCG

Genetic algorithms can be used to create an SBPCG system. The following is an outline of the steps within a GA:

1. An initial population (a collection of individuals to be tested and modified by the GA) of content pieces is saved. These may be designed manually or previously generated using an algorithm.
2. Individuals in population are evaluated using the fitness function, and assigned fitness values.
3. The selection operator selects individuals for crossover and/or mutation using their fitness values.
4. Crossover and/or mutation operators are used on selected individuals.
5. Individuals created from crossover/mutation replace lower scoring individuals in the population.
6. A function tests to see if the GA is "done". Most genetic algorithms are designed to run for a specified number of generations. This is done by comparing a counter that is incremented ever generation to the specified parameter.
7. If it is done, the population is returned. If it is not done, the process starts over from step number two.

5. APPLICATIONS

We will discuss several applications of search-based procedural content generation used to generate videogame content.

5.1 Track Generation in TORCS

The Open Racing Car Simulator (TORCS) is an open-source car racing simulator. TORCS is often used in experiments and studies due to the fact that it is open source, and because it has a detailed and complex physics engine, 3D graphics, and a variety of race tracks, cars, and game modes. Cardamone *et al.* [1] proposed a framework that uses an interactive fitness function to generate racetracks in TORCS. While the proposed system includes a single and multi-user mode for fitness testing, we will focus on results of the single-user mode.

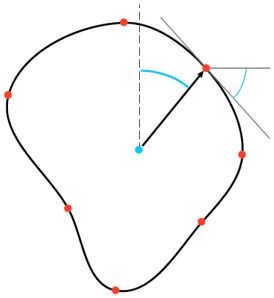

Figure 3: Race tracks are encoded using the polar coordinates of control points (the dots along the path of the track), the system origin (the dot in center), and feasible slope values for each control point (an example is represented the straight tangent line going through a control point) [1].

5.1.1 *Track Representation*

The algorithm that generates tracks takes in a number n on control points p (represented as polar coordinates). These control points form a template for a racetrack. Track segments act as experimental chunks that fill in the template, and are then generated that pass through these control points to form a closed track. The process takes place in the following way:

1. A number n of control points p (polar coordinates) are given as input to an algorithm that returns a list of track segments.
2. The polar coordinates of the n control points are used to generate a range of possible slope values for each point (see figure 3). If no slope value can be generated that would allow the track segment passing through

the control point to meet with the incoming and outgoing segments, a null track is returned. The process repeats until a closed track is generated.

3. A procedure connects the control points by using track segments that are either straight or turn segments. A straight segment is defined simply by its length as a parameter. Turn segments are defined by 4 parameters: the direction of the turn (either left or right), the arc covered by the turn represented in radians, the start radius of the turn, and the end radius of the turn.
4. If the first and last control points cannot be connected by a single track segment, a procedure uses different heuristics to attempt to connect them with a series of different segments.

Figure 4: A screenshot of the interface shows how the user can rate racetracks using the "star" (left) and "like/dislike" (right) [1].

5.1.2 Fitness Function

Within this framework, an interactive fitness test was used. The user interface presents the user with a visual display of the population of tracks. Users are able to test the tracks, then score them using one of two systems:

1. **The "star" system.** The user can rate each track from 1 to 5, represented by a number of filled in stars out of 5 total stars. A fitness value is assigned of the same integer value (i.e rating a track 2 out of 5 stars corresponds to a fitness value of 2). See Figure 4.
2. **The "like/dislike" system.** The user either likes or dislikes a track by clicking on a "thumbs up" or "thumbs down" icon, respectively. Liking a track assigns it a fitness value of 5, and disliking it assigns it a value of 1. See Figure 4.

5.1.3 Testing and Results

In the single-user mode, the framework was tested using 4 different selection operator/fitness function combinations: the star system with tournament selection, the like/dislike system with tournament selection, the star system with truncation selection, and the like/dislike system with truncation selection. Each of these was tested by 5 human subjects, who were asked to complete 10 generations, each with a population size of 20 tracks. The users were not told which selection type was being used.

After all the data was collected from the trials, the two fitness interfaces were compared (see Figure 5) and the two selection operators were compared. Average fitness scores were generally higher using the like/dislike system, while the selection operator didn't make a significant different. Users reported that they felt that liking or disliking a track was much more intuitive than assigning it a score from 1 to 5.

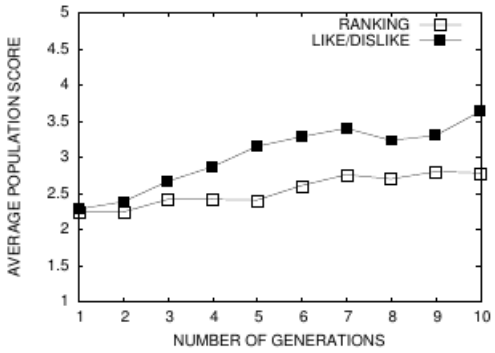

Figure 5: Average fitness scores of the population over 10 generations using the star ranking system (empty squares) and the like/dislike system (solid squares) [1].

5.2 Level Generation in a Platforming Game

In the platforming game *Prince of Persia* (1989), players must run, climb, and jump through two-dimensional levels, while avoiding traps and encountering enemies. These are all common and definitive elements of platforming games. We will next examine an offline search-based procedural content system proposed by Mourato *et al.* [2]. Their system uses a genetic algorithm to generate levels for *Prince of Persia*.

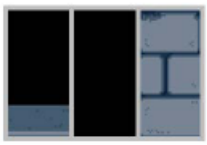

Figure 6: The three block types from left to right: floor, empty space, wall.

5.2.1 Level Representation

Each level is represented by a template in the form of a two dimensional grid of rectangular spaces, or "cells". The following are elements that make up a level.

1. **Cells.** Cells are two-dimensional rectangular tiles that make up the template of a level. Cells contain blocks, which form the structure of the level. Each level also has a starting cell where the player begins the level, and an ending cell where the player must move to in order to complete the level.
2. **Blocks.** Blocks are experimental chunks that represent pieces of geometry. Each block occupies one cell within a level template. The basic block types are walls, floors, and empty blocks. Walls create physical barriers for the player, and floors create areas that can be walked on. Empty blocks create spaces for the player to jump or fall into, or they can form gaps between platform blocks for the player to jump over.
3. **Path.** The path of a level is the area that can be traversed by the player. The area of the path is made up of both empty and floor blocks. The path in a level must also connect the starting and ending cells.

4. **Window.** Each level is divided into 10 by 3 cell sections called "windows". At any given moment during gameplay, the player is shown the window of the level that their character is currently in. When the character is moved outside of the window, the game displays the new window that the character has been moved into.
5. **Entities.** Entities are objects that are added to a level after the geometric level structure is complete. Entities include cell textures, enemies, traps, and usable items.

5.2.2 *Fitness Function*

In order to calculate the fitness scores for each level, the fitness function was designed to test aspects that would influence the way a human player perceives the level. Fitness values assigned to levels range from 0 to 1. A fitness value of around 0.85 corresponds to a level that typically has no significant flaws, and presents the player with an adequate challenge. The fitness function is direct, and tests the following elements individually to calculate late a score for each level:

1. **Path Structure.** The fitness function favors paths that would, theoretically, increase the players' immersion in the game. The function looks at the complexity of the path structure, and favors paths that are non-linear, but not too complex. Non-linear paths that include multiple ways to reach the ending cell may be more fun to players, but paths with excessive branches and dead ends may seem overwhelming to the player. The function also examines the minimum number of moves (moving in some direction, jumping, climbing) that a player must make to traverse the entire area of the path. This is brief measurement of the difficulty of a level.
2. **Individual Cells.** Individual cells are examined to see if they make sense in the context of the other cells in the level. The function looks for "valid" cells, meaning the blocks within them can be used in a meaningful way. For example, a cell that contains a floor block is valid if it is part of the path. A cell with a floor block surrounded entirely by wall blocks would be non-reachable by the player, and therefore invalid. The same rule applies to cells containing empty blocks. Cells containing wall blocks are always valid. The "value" of a cell refers to the block that is occupying it.
3. **Ending.** The placement of the ending cell is examined to ensure the area of the path that the player must navigate to reach it is sufficiently challenging.
4. **Balance.** The function also checks for a balance between the types of blocks that are used to make up the level. The number of each type of block used to create a level should be fairly close.
5. **Space Usage.** The fitness function also examines the number of cells in a levels path, in relation to the total number of cells in the level.

5.2.3 *Genetic Operators*

The following are the genetic operators used in this system:

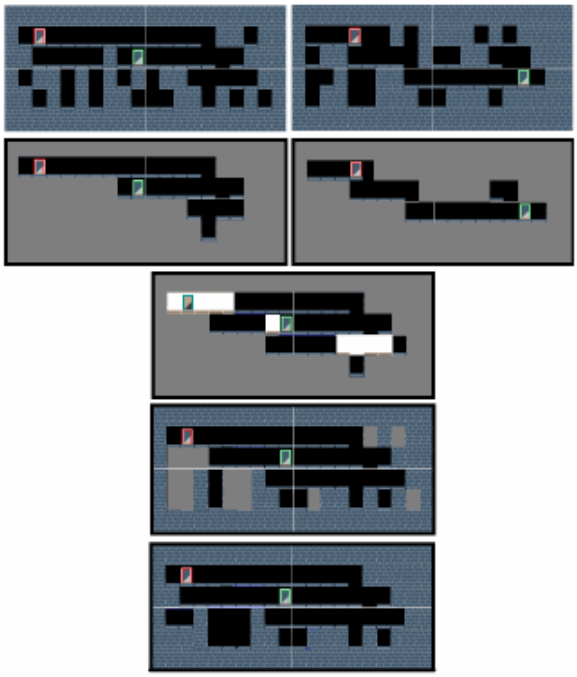

Figure 7: An example of the crossover operation. Each row represents a different step in the process. See section 5.2.3 [2].

1. **Crossover.** Crossover is performed on two levels with the following steps:
   (a) The paths of each level are combined (cells in both original paths are placed in their corresponding locations within a new level) to form one path in a new level. See figure 7, second and third rows.
   (b) Overlapping path cells (cells that occupy the same grid position in both original level paths) within the new level are assigned their corresponding values from one of the original levels. See figure 7, third row.
   (c) Other cells of the same values in the original levels are added to the new level. See figure 7, fourth row.
   (d) The remaining cells in the new level take on corresponding values from either of the two (one is picked at random) original levels. See figure 7, fifth row.
2. **Mutation.** In this system, mutation works by changing the values of several cells. The specific number of cells to be changed during mutation can be adjusted, as it is a system parameter. Two forms of mutation were implemented:
   (a) **Random Mutation.** Random mutation works by simply picking random cells within a level to change.
   (b) **Selective Mutation.** Selective mutation makes specific types of mutation more likely to occur. For example, cells that are not part of the path,

and therefore not accessible to the player, are more likely to be changed to wall cells.

5.2.4 Testing and Results

The system was tested using a prototype program that allowed parameters (number cell columns and rows, number of individual levels in the population, and the number of generation) to be adjusted through a graphical user interface. Testing was performed using a level grid of 4 by 5 windows (40 by 15 cells), as this corresponds to a reasonably sized level from the original game. Population sizes of 20, 50, 100, and 200 individuals were each tested with 200, 500, 1000, and 2000 generations. Each combination was tested 20 times. Average fitness values and time (in seconds) were recorded. See Figure 8.

Generations Population	200	500	1000	2000
20 individuals	$\mu_t = 0.5$ $\sigma_t = 0.1$ $\mu_f = 0.72$ $\sigma_f = 0.04$	$\mu_t = 1.7$ $\sigma_t = 0.6$ $\mu_f = 0.77$ $\sigma_f = 0.06$	$\mu_t = 4.5$ $\sigma_t = 1.9$ $\mu_f = 0.82$ $\sigma_f = 0.05$	$\mu_t = 13$ $\sigma_t = 4.5$ $\mu_f = 0.87$ $\sigma_f = 0.04$
50 individuals	$\mu_t = 1.6$ $\sigma_t = 0.4$ $\mu_f = 0.76$ $\sigma_f = 0.03$	$\mu_t = 5.6$ $\sigma_t = 3.3$ $\mu_f = 0.85$ $\sigma_f = 0.04$	$\mu_t = 14$ $\sigma_t = 5.4$ $\mu_f = 0.86$ $\sigma_f = 0.04$	$\mu_t = 26$ $\sigma_t = 10$ $\mu_f = 0.89$ $\sigma_f = 0.03$
100 individuals	$\mu_t = 3.3$ $\sigma_t = 1$ $\mu_f = 0.81$ $\sigma_f = 0.05$	$\mu_t = 9.5$ $\sigma_t = 3.3$ $\mu_f = 0.84$ $\sigma_f = 0.06$	$\mu_t = 19$ $\sigma_t = 3.8$ $\mu_f = 0.89$ $\sigma_f = 0.06$	$\mu_t = 51$ $\sigma_t = 22.5$ $\mu_f = 0.92$ $\sigma_f = 0.04$
200 individuals	$\mu_t = 7.5$ $\sigma_t = 3$ $\mu_f = 0.83$ $\sigma_f = 0.07$	$\mu_t = 23$ $\sigma_t = 12$ $\mu_f = 0.87$ $\sigma_f = 0.05$	$\mu_t = 46$ $\sigma_t = 17$ $\mu_f = 0.92$ $\sigma_f = 0.05$	$\mu_t = 102$ $\sigma_t = 39$ $\mu_f = 0.93$ $\sigma_f = 0.06$

Figure 8: This table displays results for each combination rested. Averages μ and standard deviations σ are displayed for computation times t (in seconds), and fitness values f [2].

Overall, an increase in average fitness values can be seen when the number of generations increased, and when larger population sizes were used. As expected, the computation time also increased when computing more generations and using larger population sizes. While computation times will vary depending on the hardware used, the worst case in the testing results (200 individuals, 2000 generations) yielded an average fitness value of 0.93 (reasonably higher than the goal of 0.85), with an average time of 102 seconds.

6. CONCLUSIONS

Both systems discussed in this paper show how search-based PCG systems can improve the quality of generated content. They do, however, achieve this in different ways. The system proposed by Cardamone *et al.* [1] attempted to improve procedurally generated racetracks by implementing an SBPCG system with an interactive fitness function. The data may not show any particularly dramatic increase in fitness values, however, it does directly reflect the users' enjoyment of the content, as fitness values were calculated from user ratings. The system is still in its preliminary stages, but the data and feedback collected suggests interactive fitness functions can be very effective for determining the quality of procedurally generated racetracks.

The system proposed by Mourato *et al.* [2] for generating levels in *Prince of Persia* also yielded impressive results. While testing the system with 200 generations of population of 50 individuals, an average fitness value of 0.85 (the desired fitness value for a level) was achieved with an average time of 5.6 seconds. Even the worst case that was tested (2000 generations, 200 individuals) performed reasonably well, generating a a population with an average fitness value of 0.93 in an average time of 102 seconds. The proposed system seems to be an effective solution for procedural level generation in *Prince of Persia* and other games that use two-dimensional, grid-based level design.

7. ACKNOWLEDGMENTS

I would like to thank Elena Machkasova for assisting me as both the course professor, and my adviser. I would also like to thank Wayne Manselle for giving me thoughtful and helpful feedback.

8. REFERENCES

[1] L. Cardamone, D. Loiacono, and P. L. Lanzi. 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*, GECCO '11, pages 395–402, New York, NY, USA, 2011. ACM.

[2] F. Mourato, M. P. dos Santos, and F. Birra. Automatic level generation for platform videogames using genetic algorithms. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology*, ACE '11, pages 8:1–8:8, New York, NY, USA, 2011. ACM.

[3] M. Obitko. Introduction to genetic algorithms, 1998.

[4] G. Smith. Understanding procedural content generation: A design-centric analysis of the role of pcg in games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 917–926, New York, NY, USA, 2014. ACM.

[5] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. Search-based procedural content generation. In *Proceedings of the 2010 International Conference on Applications of Evolutionary Computation - Volume Part I*, EvoApplicatons'10, pages 141–150, Berlin, Heidelberg, 2010. Springer-Verlag.

[6] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey, 2011.

[7] Wikipedia. Genetic operator — Wikipedia, the free encyclopedia, 2014.

[8] Wikipedia. Selection (genetic algorithm) — Wikipedia, the free encyclopedia, 2014.