Procedural Content Generation: Techniques and Applications

Richard Stangl
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
stang149@morris.umn.edu

ABSTRACT

Procedural Content Generation, or PCG, is an established method of using algorithmic systems to create video game assets and content ranging from simple image textures to entire environments. There are nearly as many implementations of PCG as there are games that use it but attempts have been made to create a general framework to allow a PCG implementation to be reused for multiple games. Additionally, evolutionary computation and genetic programming are often central to modern PCG systems and can be used to create game assets such as densely packed dungeon maps.

Keywords

ACM proceedings, Procedural Content Generation, PCG, Procedural Generation

1. INTRODUCTION

Procedural Content Generation is a common, powerful tool used by developers to both enhance their own efforts during the development period of a game as well as a central component of gameplay where it is used to create seemingly endless game levels or even entire game worlds that stretch as far as the player explores. Evolutionary computation and genetic programming are often the backbone of these methods serving to both generate the variety of content needed and to tailor the content to the developer's goals. Two implementations of PCG are described here to demonstrate how these powerful algorithms can generate large portions of a completed game.

2. BACKGROUND

Procedural Content Generation in Games defines PCG as follows: "PCG is the algorithmic creation of game content with limited or indirect user input." [4] In practice, PCG is the use of algorithms to generate one or more aspects of a game. In experimental cases entire games can be built from the ground using PCG to create everything from rules and structure to content and assets. In their definition "content" refers to "most of what is contained in a game: levels, maps, game rules, textures, stories, items, quests, music, weapons, vehicles, characters, etc." Their definition specifically excludes the game engine and non-player character (NPC) artificial intelligence and behavior. In practice, PCG methods are specific to each game they are designed for; there has been little movement towards a more general PCG method that could be reused for multiple games.

PCG was originally conceived as a method to overcome storage limitations in the 1980s. Early games like *Elite* (1984) and *Rogue* (1980) built PCG systems into the game code that used seed numbers to generate game content. However, once storage was no longer a limiting factor, game designers began to use PCG as a method of creating unique and/or endless content to enhance a gaming experience. Examples of popular games that utilize PCG range from those that generate relatively simple content such as game levels like *Tiny Wings* or *Spelunky* to those that base the whole game experience around PCG such as *Minecraft* (2011) and *Dwarf Fortress* (2006), where nearly all aspects of the game are procedurally generated.

There are several reasons a game designer might decide to use PCG. The first and perhaps most straightforward reason is to reduce the amount of human effort required to create a game. As major tentpole games continue to become more elaborate and more expensive to produce PCG, can be used as a method for cost saving. A similar reason to use PCG occurs at the opposite end of the market spectrum. Small independent developers may not have the experience necessary to create certain aspects of a game manually or may not have the resources to do so. PCG can supplement a small team to create a more elaborate game than they might have otherwise. Additionally, PCG can be used to stimulate creativity. The automated, algorithmic process often generates ideas that designers may never have come up with on their own. PCG can also be used to create new kinds of games such as games that generate content as it is consumed to extend the length of a user experience. This is comparable to the "endless runner" genre of mobile games however on a much grander scale. PCG is also often used to tailor game content for individual users, whether by taking gamer input into account or by automatically adjusting a game's difficulty to take a user's skill level into account. [3]

There are a few properties to take into account when designing a PCG method as laid out in *Procedural Content Generation in Games.* The first and perhaps most broad is speed. PCG is often used either during the design process or during gameplay itself and each situation has different constraints for the speed of the PCG method. During the design process a PCG method may take days or months to reach

UMM CSci Senior Seminar Conference, May 2017 Morris, MN.

a solution without adversely affecting game development, but content generation during gameplay must be nearly instantaneous so as to not affect the gaming experience. A PCG method must also have the reliability needed for the given situation. A method that generates a complete level design is not effective if the level is unwinnable. However, if a method is used to generate vegetation for a game environment it is less crucial that every tree look perfect. PCG methods must often be controllable so a developer or user can have some input in the type of content generated, e.g. the length of a level or the number of enemies, etc. Expressivity and diversity must also be taken into account. Generated content would hardly be worthwhile if it was repetitive or boring. Additionally, content should be creative and believable. Content that is obviously algorithmically generated may take users out of the gaming experience. [4] [1]

3. VIDEO GAME LEVEL GENERATION

At a high level, one goal of PCG is a system that generates game levels for multiple games. This would allow developers to reuse a PCG tool much like game engines or 3D models are reused for similar games. However, there are two main limiting factors to this approach. Firstly, it is simply unknown how to create competent level generators for more than one game with current approaches and methods. The second factor is the sheer amount of work needed to create a single level generator. The second factor is largely responsible for the first; it has been impractical to develop level generators that work for multiple games since these generators would necessarily need to be hybrids of major portions of several individual game level generators and each individual generator requires significant resource investment. This problem has been looked at from a theoretical or academic perspective as general level generation much like how General Game Playing and General Video Game Playing have been seen as theoretical artificial intelligence (AI) problems. The definition of the problem of general level generation as given by Khalifa et all is put as follows: "Construct a generator that, given a game described in a specific description language and which can be played by some AI player, builds any required number of different levels for that game which are enjoyable for humans to play." [2]

3.1 General Video Game Level Generation

One attempt at generic level generation is described in General Video Game Level Generation (GVG-LG). The authors describe a Java framework built on top of the General Video Game Artificial Intelligence (GVG-AI) framework which they have named General Video Game Level Generation (GVG-LG). GVG-AI was built for the 2014 General Video Game Playing Competition for developers to design a generic game-playing AI. Games and controllers for the competition can be reused for GVG-LG, which is why the authors chose this framework to build on. Essentially, while GVG-AI took a game player AI as an input for the competition, GVG-LG allows a developer to plug in various level generators and test them within the framework. [2]

The game descriptions used in the GVG-AI are made up of four elements: sprites, termination conditions, interactions, and level mapping. Sprites are the main objects and come in six varieties: Avatar, NPC, Resource, Portal, Static, and Moving. Each is represented as a SpriteData data structure containing its name and information. TerminationData are similar in construction and contain information on the termination conditions: type, a limit, a win flag, and a list of sprite names. InteractionData contain instructions on what should happen in the case of a collision between two objects. Level mapping is stored as a HashMap (LevelMapping) which maps the sprites by name. The current generator can retrieve all this information and replace the level map through the getLevelMapping function. Generated levels must fulfill two basic conditions: the level may not have more than one Avatar and cannot contain anything not in the original or new LevelMapping.

3.2 Included Generators

The GVG-AI framework is built to take game level generators as an input for testing. The framework is meant for testing user created generators by analyzing the levels they generate, however the framework also includes sample generators. These are described in the next section.

3.2.1 Random Level Generator

The first and simplest of the level generators in the GVG-LG is a random level generator. It generates position on the room map and type of object based on an adjustable probability. This generator will follow the basic conditions as well as include at least one of each type of sprite. An example of a small room generated by this process is pictured in Figure 2a. The squares around the perimeter of the room and the three similar squares in the lower half of the room are static sprites (walls and obstacles). There are three room exits in this room and the sprite near the bottom two is the avatar sprite. The remaining sprites are enemies and a key. The sprites seem randomly placed apart from the perimeter wall as would be expected of a random placement algorithm.

3.2.2 Constructive Level Generator

This generator analyzes the GameDescription object (which contains all of the game objects) and first classifies all current sprites into the following categories: Avatar Sprites, Solid Sprites (obstacles/walls), Harmful Sprites, Collectible Sprites (destroyed during interaction with player), and Other Sprites. It then moves on to the main generation procedure, which takes place over four main steps and pre- and post-processing.

During the pre-processing step the generator calculates how many tiles of the generated level should contain sprites and at what percentage each type of sprite should be present. If there is a large percentage of collectible sprites, the total sprite count will go up, but will decrease if there is a large percentage of harmful sprites.

The generator then moves on to the main event: sprite placement. Figure 1 illustrates each of the four steps of sprite placement. The first step is to build a level layout. The level is first surrounded by solid sprites as a sort of world boundary, then it is filled with the correct number of solid sprites as walls or obstacles. The generator makes sure to not block off any area of the map. An avatar sprite is placed in a random open tile. Based on the attributes of the avatar the location may be limited to certain areas. For example, if the avatar can only move horizontally as in a game like Space Invaders the avatar sprite will be placed at either the top or bottom of the level. Harmful sprites are added next. Mobile sprites will be placed at a distance from the avatar while stationary sprites (such as spikes or thorns)

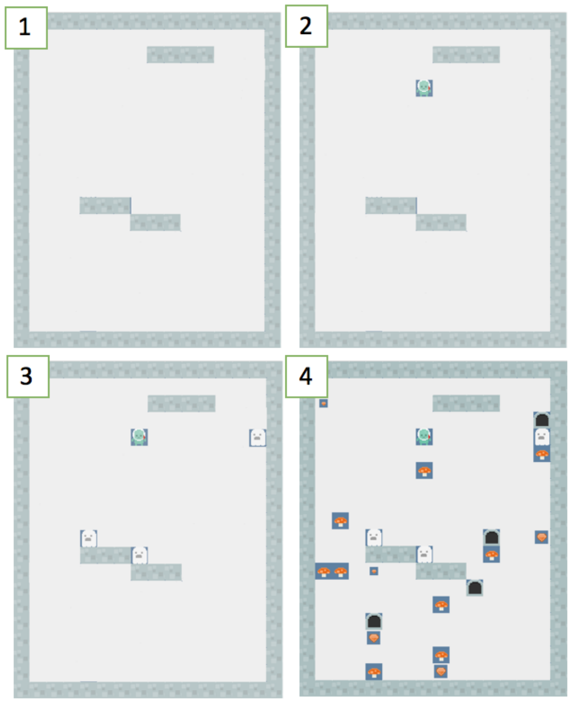

Figure 1: The four steps of sprite placement [2]

are placed in any open tile. The remaining sprites are added at random locations based on the correct ratios defined in the pre-processing step.

Finally, the generator moves on to post-processing which consists of fixing goal sprites. The generator makes sure that there are more goal sprites than the number specified for game termination. More are added if needed. For instance, if the goal is to defeat ten enemies, the generator would make sure there are at least ten mobile hostile sprites. Figure 2b depicts a small example of a room generated via this generator. There is more order and purpose in the placement of sprites in this room but it lacks complexity.

3.2.3 Search-based Level Generator

The search-based level generator uses a genetic algorithm called Feasible Infeasible 2 Population Genetic Algorithm (FI2Pop) to evolve two populations, feasible and infeasible, of initial game states. The feasible population focuses on improving game states to fit the desired outcome, while the infeasible population is meant to remove game levels that violate the problem constraints. Genetic algorithms such as FI2Pop are a form of evolutionary computation (EC), which uses evolutionary principles to gradually shape a population of programs or program objects to a desired outcome. In this case the genetic algorithm is evolving game states. In EC a beginning population that fits an initial condition is created. Members of the population are then mutated by functions that may add, delete, or modify portions of each member. Often two members (or parents) are combined in some way to create a new population member (or child). The resulting population is then evaluated by a heuristic function. A heuristic function could be described as a filter that selects among the children of a genetic population for those with outcomes that move the problem closer to the solution. This process is repeated until a child matches a desired outcome or, in the case of an unsuccessful attempt, until the process is repeated more times than allowed by the programmer. Each initial game state is generated using the constructive level generator, and each successive population is created by mutating members of the parent generation. These mutations either add a random sprite to a random tile, remove sprites from a random tile, or swap sprites between two random tiles. Child initial game states can transfer between the two populations, but each population evolves separately. Child initial game states are evaluated using three player AI that were developed for the gameplay competition. The first and standard controller (the term used for these AI) is a modified version of the winner of the 2014 competition, Adrienctx. The controller was modified to increase reaction time to make it more human-like. The results of the Adrienctx playthrough are compared to two simple controllers, OneStepLookAhead, in which the controller chooses a beneficial next step from the available options, and DoNothing, in which the controller does nothing each step.

There are two methods of evaluating the feasible population children using the three controllers. Each is part of a heuristic function that selects the best candidates for mutation. The Score Difference Fitness is the difference between the Adrienctx score and the best score by OneStepLookAhead after 50 runs. This method is designed to generate levels that require skill for a higher score. The OneStepLookAhead controller is meant to simulate a poor player so the greater the difference the greater the skill differential. The second method generates a score based on the number of unique events that happened over the level playthrough in order to favor levels that require the player to use more of the game rules in the assumption that a good level doesn't rely on only one or two game mechanics. The final score used in the heuristic function is an average of the scores from the two methods in an effort to balance difficulty level against novelty and variety in the game levels.

The infeasible population is simply constrained by seven factors. Each game level must have one avatar and must have one or more of each type of sprite (not including sprites that spawn from other sprites such as arrows). There must be more goal sprites than the terminating factor requires, i.e. if the game should have more than zero enemies at any time there must always be at least one enemy. Between 5-30% of the game tiles must have sprites on them. Game levels must not be solved by any controller in less than 200 steps and the Adrienctx controller must win the game level during the evaluation by the heuristic function. Also, the DoNothing controller must not be killed in the first 40 steps in 50 different runs and it cannot win in the same number of steps as Adrienctx. These last rules are to guarantee that the game is not too hard in the beginning and is not so easy that it is beatable early in the game. Figure 2c shows a search-based level generated room. There is even more structure and appearance of design intent than was present in the constructive level generated room.

4. RESULTS

A small study was done using human players to gauge preference between the three generators. Each generator was used to create five levels for each of three different simple games. These games were based on *Frogger*, *Pac-Man*, and *The Legend of Zelda*. While the authors expected the

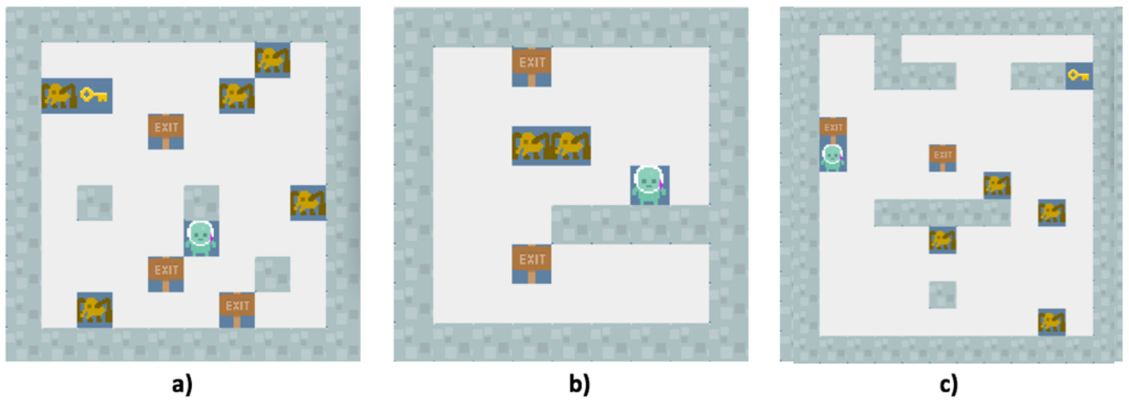

Figure 2: Levels generated using a) random level generation, b) constructive level generation, and c) search-based level generator. [2]

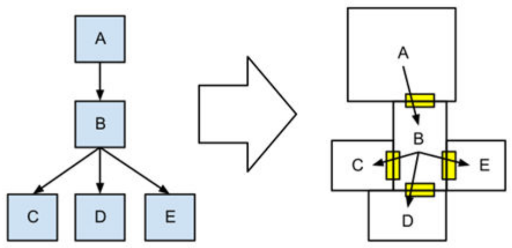

Figure 3: The translation of a tree structure to a room layout [5]

players to prefer the search-based generated levels to the constructive levels and to prefer both to the randomly generated levels, the players were unable to distinguish between the constructive and randomly generated levels but a clear preference was shown for the search-based generated levels.

5. VIDEO GAME LEVEL EVOLUTION

Section 3.2.3 discussed a procedural content generator that used a simple genetic algorithm to improve level design. This section will discuss a PCG that uses a more advanced genetic program as described in Evolving Dungeon Crawler Levels With Relative Placement. Instead of placing sprites on tiles, this map generation creates a series of connected rooms.

5.1 Element Representation

Each member of the evolutionary population is represented by a tree structure as depicted in Figure 3. Every node represents a room and is connected to both its parent and its children by a door. The tree is processed breadth-first (top to bottom) meaning that the root node becomes the first room and its children are adjoining rooms. Each node has a tile property that sets the location or absence of the doors and the size and shape of the room. The tile property also stores the room's type, which can either be "hallway," "event," or none/regular. After placing the parent room, the translation process attempts to connect each of its children to the parent room in order left to right. If a child cannot fit in the space provided based on the dimensions of nearby rooms or there are no more doors available that child and all its child nodes are deleted. After each of these children are processed the algorithm moves to the next node (top to bottom, left to right). If a new room is placed that happens to lie next to a previous room and both rooms have available doors, the two rooms are connected. After the tree has been traversed completely a shortest weighted path is generated from the root room to each child room. If a room's shortest path exceeds the allowed distance, that room and its children are removed. This culling prevents further development along that node path which prevents inefficiency by removing distant rooms.

5.2 Operators

There are three main operators that are used to evolve each tree. The first is the crossover operator which copies a random sub-tree and connects it to another parent node. This process allows for a subtree to possibly find a better location on the tree. It also generates a certain amount of symmetry across the tree which gives the illusion of human design intent.

The second operator is the mutation operator which is made up of three subroutines: mutation_Grow, mutation_Trim, and mutation_Change. mutation_Grow selects x random rooms with an unused door (doors only exist on the map if they connect two rooms) and adds y new nodes to them. The new rooms are given random attributes. mutation_Trim works largely in reverse. It finds a random room and removes a leaf node or, if the room has no leaves, deletes the room. mutation_Change picks x random rooms and gives them new shapes and numbers of doors.

The last operator divides the population into a sort of tournament. This algorithm sorts the rooms using the fitness function described later. The rooms are sorted by score and the bottom half of the population is replaced with children of the top half. These new rooms are created by using the other two operators on the higher scoring rooms.

5.3 Fitness Heuristic

The heuristic used in this generator takes two factors into account: it prefers rooms made up of tightly packed clusters

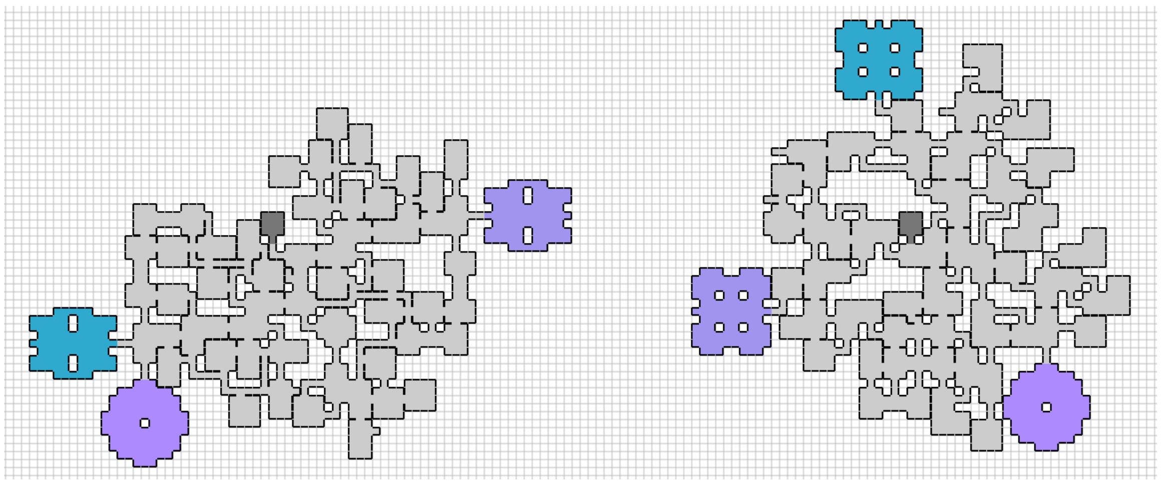

Figure 4: Maps generated for Fitness Impact Experiment (left) and On-Demand Generation Experiment (right) [5]

of rooms connected to efficient hallways and rejects maps that contain more than three large, special rooms meant for certain game events. Hallways and the special event rooms are regular rooms generated by the tree that are given these special attributes during creation. The event rooms are much larger than other rooms and the heuristic prefers them to be near the outside edges of the map. Two examples of ideal map layouts are shown in Figure 4.

To begin evaluation, all hallway rooms that are connected to each other are combined into one hallway. Each new hallway is awarded a score based on how many non-hallway rooms are connected to each other via the hallway. Hallways that connect no rooms together, i.e. those that resemble ordinary rooms, are not given a score. A small penalty is given to each new hallway based on the number of original hallway rooms make them up. This penalty is meant to encourage efficient hallways. Additionally, the fitness heuristic rewards normal rooms that are connected to both a hallway and at least one other regular room in order to favor tight clusters of rooms centered around the hallways.

The process for favoring maps that contain between one and three event rooms is relatively simple. Maps are awarded a high score for each event room it contains but the score is reset to zero if a map contains more than three event rooms. Maps that have event rooms on the outer edges are favored by only awarding maps event room bonuses if the event rooms are more than a required minimum distance from the origin room.

5.4 Evaluation

Valtchanov and Brown [5] set up several experiments to test the effectiveness of the fitness heuristic. Two methods used are detailed here. Figure 4 depicts a resulting map from each of these methods of evaluation.

5.4.1 *Fitness Impact Experiment*

This experiment was designed to test how much the general structure of the map is determined by the fitness heuristic. The generator was allowed to run for a excessively long time (2000 generations) and a limit on rooms that exceeded what it would ever want to create (500 rooms). There was no significant increase in score beyond 1500 generations and the room limit was never reached.

The resulting maps were found to greatly resemble the types of maps the heuristic was designed to favor. Every map contained three event rooms and long hallways with many room connections were common. There were also many rooms that were connected to both hallways and regular rooms. The maps were also tightly packed and made good use of space. Additionally, the rooms varied greatly in structure, showing that the heuristic did not guide development towards a single type of map. In other words, running the generator for an excessive number of generations only encouraged maps to better fit the heuristic rather than pushing them towards a specific map that happened to fit the heuristic.

5.4.2 *On-Demand Generation Experiment*

The second experiment tested how well the generator performed under constrained situations. If the generator performed well it would be suited to generate maps as required by a player. This type of generator could be used for games that create content dynamically rather than just included only pre-generated maps. The generator was only allowed to run for 500 generations and the maps were capped at only 100 rooms. The resulting maps took 30 seconds to generate using a single core of a 2.4 GHz processor. These maps were impressively similar to the maps created under the ideal situations in the Fitness Impact Experiment which supports the idea that the fitness heuristic can help the generator create nearly ideal maps in less than ideal conditions. This similarity can be seen by comparing the maps in Figure 4.

6. CONCLUSIONS

The results of the two procedural content generators described here illustrate the promising future this game development tool has in the game industry. A game generator made up of a hybrid of these two PCG implementations, where GVG-LG is used to create the layout of sprites in each of the rooms in a map created by Valtchanov and Brown's generator, could be used to create a large percentage of a completed game with minimal effort on the developer's part.

The game industry is largely bifurcating either into tentpole games that require teams of developers and months of work or small indie games made by just a few developers. Tentpole games such as *No Man's Sky* and indie games like *Minecraft* have gameplay elements that are largely based around PCG. In both game design environments PCG can be used to improve the output of both these groups, and that outcome is good for developers and gamers alike.

Acknowledgments

I would like to thank Elena Machkasova for her instruction on this project. I would also like to thank KK Lamberty for her feedback throughout the writing process. Lastly, I would like to thank Wayne Manselle for his review and critique of this manuscript.

7. REFERENCES

[1] J. Dormans. Adventures in level design: Generating missions and spaces for action adventure games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 1:1–1:8, New York, NY, USA, 2010. ACM.

[2] A. Khalifa, D. Perez-Liebana, S. M. Lucas, and J. Togelius. General video game level generation. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 253–259, New York, NY, USA, 2016. ACM.

[3] J. Togelius, E. Kastbjerg, D. Schedl, and G. N. Yannakakis. What is procedural content generation?: Mario on the borderline. In *Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games*, PCGames '11, pages 3:1–3:6, New York, NY, USA, 2011. ACM.

[4] J. Togelius, N. Shaker, and M. J. Nelson. Introduction. In N. Shaker, J. Togelius, and M. J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, pages 1–15. Springer, 2016.

[5] V. Valtchanov and J. A. Brown. Evolving dungeon crawler levels with relative placement. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, C3S2E '12, pages 27–35, New York, NY, USA, 2012. ACM.