Automating Dungeon Level Design using Search-based Procedural Generation

Arthur R. Pinheiro So[1], Alinne C. Corrêa Souza[1], Lincoln M. Costa[2], Rafael G. Mantovani[3], Francisco Carlos M. Souz[1]

[1]Federal University of Technology – Paraná – Dois Vizinhos, PR, Brasil

[2]Federal University of Rio de Janeiro (UFRJ) – Rio de janeiro, RJ – Brasil

[3]Federal University of Technology – Paraná – Apucarana, PR, Brasil

arthurriuiti@alunos.utfpr.edu.br, costa@cos.ufrj.br, rgmantovani@gmail.com,

{alinnesouza, franciscosouza}@utfpr.edu.br

Abstract. *The adoption of procedural content generation in electronic games is increasing, allowing for the quick and cost-effective creation of game artifacts like music, scenarios, and art, which aids or replaces manual content creation. Metaheuristics, such as Genetic Algorithms are effective in solving complex problems by exploring the solution space. This study implemented and analyzed this algorithm for generating procedural content in dungeon games, focusing on scenarios and enemies. Experimental evaluations demonstrated efficient optimization of room layouts and enemy distributions, with smaller room sizes producing better fitness results.*

Keywords *procedural content generation, genetic algorithm, 2d game design*

1. Introduction

As games get increasingly complex, the time and effort required to produce content manually increases (Adams 2009), and the estimated time to produce is, on average, three years or more. A significant part of this time is dedicated to creating item models, maps, and avatars, among other artifacts. Not only is it hard work, but also a time-consuming and costly process. In this context, solutions are needed to minimize the cost and time of production of these artifacts. A method to reduce costs is Procedural Content Generation (PCG), which automatically creates content using different algorithms (TOGELIUS et al. 2011). The application of procedural generation techniques quickly allows the creation of a large volume of content that can be used within the game, i.e., this content is randomly generated several times, serving as a starting point for the developer. It is important to note that in addition to PCG techniques, it is possible to employ Artificial Intelligence (AI) such as meta-heuristics to assist in this process. Meta-heuristics are search algorithms that aim to find optimal solutions to complex problems; among the various techniques, it is possible to highlight the Genetic Algorithm (GA) (Russel e Norvig 2013).

In this context, this paper contributes to the area PCG with a level generator that uses two GAs to create dungeons and to allocate and balance enemies within based on their attributes such as strength and health points. The remainder of this paper is organized as follows: Section 2 provides related works. Section 3 describes the proposed methods.

Section 4 presents and discuss the results. Section 5 concludes this research paper and suggests future work.

2. Related Work

The literature on Procedural Content Generation (PCG) in games shows diverse approaches and focuses. De Lima (de Lima et al. 2019) uses genetic algorithms (GA) to generate game quests, combining planning with an evolutionary search strategy guided by story arcs, producing quests comparable to those by professional designers. Brown's work (Brown et al. 2017) focuses on the action game Hotline Miami, which requires quick reflexes and features more minor, frantic levels. The study highlights the use of algorithms in the game's level editor, integrating PCG within the commercial game. Ruela and Guimarães(Ruela e Guimarães 2014) use PCG to enhance strategies in the MMORTS game Call of Roma. The game involves allocating various soldiers under Heroes, each with specific attributes. They employ a cooperative co-evolution algorithm, where multiple genetic algorithms evolve their populations in parallel and cooperatively. In their experiments, each of the N Heroes has a corresponding sub-population that evolves independently until evaluated together during cooperative intervals.

3. PROPOSED METHODS

In this section, we detail the experimental methodology adopted in this article. An overview of the flow of experiments, including sub-steps, is shown in Figure 1. The following sub-sections give additional details regarding them.

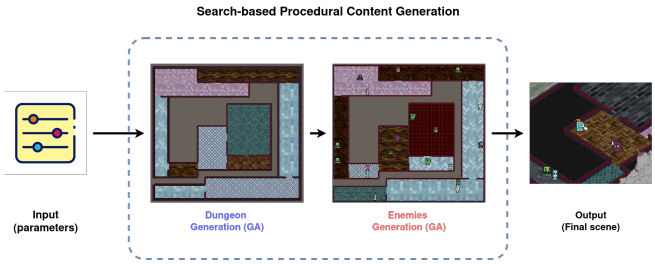


Figure 1. Pipeline defined for the PCG using meta-heuristics.

Dungeon Generation. The GA implementation for room positioning is the same as in Brown et al.'s study (Brown et al. 2017), which showed accurate results for positioning rooms in the game Hotline Miami. This layout can also be used for generic dungeons, making it suitable for the proposed game scenarios. In the GA chromosome, the dungeon layout is represented by a vector of rooms, each with attributes of position, width, height, and placement type. The placement type can be "T" for rooms that override other room areas or "U" for rooms that only occupy empty spaces in the grid.

The algorithm takes as input N_{rooms}, representing the number of rooms it will attempt to place in the grid and the grid's size (width and height). Initially, the rooms have their attributes randomly defined. The algorithm then sequentially allocates these rooms within the grid according to width, height, and position. The placement type attribute determines whether a room is placed above or below other previously placed rooms. A room will be excluded from placement if it ends up disconnected from different rooms or overlaps or splits any previously placed room. An equation proposed by Brown

et al. (Brown et al. 2017) was adopted to measure the fitness of the GA individuals. This equation considers the attributes of the information and assigns a fitness value to indicate the development level of the generated scenario. Equation enhances fitness based on the number of rooms positioned, avoiding narrow corridors and tiny quadrants. It also positively favors a large graph diameter and an average graph degree close to 2, promoting sequentially interconnected rooms and avoiding excessive interconnections. After creating the initial population, the GA combines individuals with the highest fitness, with even rooms inherited from one parent and odd rooms from another. Some individuals may undergo mutation to introduce variation. The GA results in a dungeon with maximized fitness, which will be used in the game.

Enemies Generation. After generating the dungeon layout, the second task is to create the enemies, which are essential for gameplay as they provide the main opposition for the player. A well-designed dungeon features enemies dispersed throughout its rooms, each with a balanced mix of enemy attributes. Powerful enemies can deter less experienced players, while weak enemies might need to provide more challenge. The dungeon generation is based on balancing enemies' attributes, with each enemy having values from 0 to 100. Powerful enemies have higher attribute values and more significant weight in the dungeon's balance. Enemies in the same room must be balanced to avoid having the most robust set of enemies in one room. The challenge is distributing randomly generated enemies among dungeon rooms, with enemy values corresponding to the sum of their attributes.

Initially, a list of N enemies with random attributes (*Health, Damage, Velocity, and Attack Cooldown*) is generated, forming the *EnemyPool* for allocation inside dungeon rooms. The chromosome of an individual is a list representing the room allocation for each enemy. The size of the N list represents the maximum number of enemies in the layout. Initially, some values are set to random rooms, with the rest null, meaning those enemies are not allocated. Mutations alter some genes to allocate or de-allocate enemies, while the crossover operator randomly exchanges genes between two individuals. The fitness calculation favors approximating the room totals to a specific value called $target$: in this study, this value is set to 400. It was calculated based on the ideal balanced room in which all enemies are balanced with 50 out of 4 attributes, with the total sum of attributes being 200 per enemy, and each room in this context has 2 enemies, totaling 400 points in the total of a room's attributes. The variable $target = 400$ is applied in equations (1):

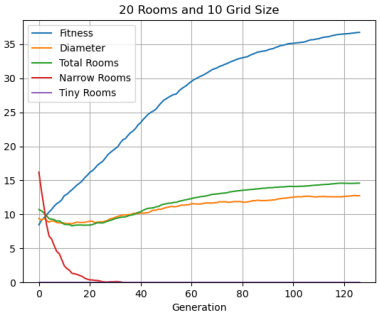
$$Fitness = \frac{|(N_{room} * target) - totDiffSum|}{N_{room} * target} \tag{1}$$

Where the total difference sum (totDiffSum) is calculated as the sum of the absolute differences between the target value and each accumulated sum (attSum) for all rooms, from room 0 to room N_{room} and $attSum_n$ is the sum of attributes of room n and N_{room} corresponds to the number of rooms in the dungeon.

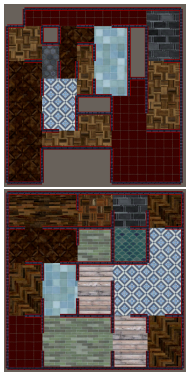
4. RESULTS

Expressiveness of Layout Generation. The results of the experiment carried out with the setup defined by 20 rooms and a 10x10 grid are presented in Figure 2. A behavior observed in the experiments is the rapid optimization of the number of rooms since the

variable N_{rooms} is the one that most positively impacts the final fitness. We observed a prominence in the experiment performed with ten rooms and 20 grid sizes. The diameter of the graph and $expDegree$ variable are optimized next. The $expDegree$ tends to reach 1, after which it stops increasing, and the diameter significantly impacts the final fitness. We also observed that during the initial generations of some experiments, there is a decrease in values that positively impact fitness, prioritizing the reduction of N_{narrow} and N_{tiny} (narrow and small rooms). These values have a very negative impact on the construction of the dungeon. This behavior is particularly evident in the experiment with 20 rooms and ten sizes (as shown in Figure 2), where the parameters significantly limit the positioning of the rooms.



((a)) Fitness and other attributes evolution



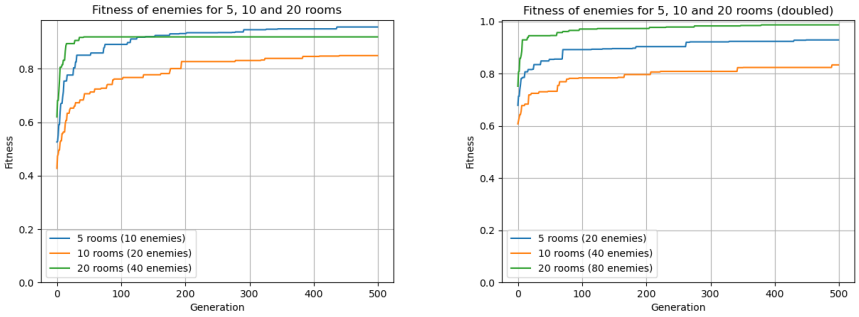
((b)) Layout before and after the optimization

Figure 2. Level generated with 20 rooms in a 10x10 grid

Enemy Generation Expressiveness. Figure 3 shows the average results for the standard and double *EnemyPool* sizes. Through GA executions, we verified that the solution converges after 500 generations with 500 individuals in the population, regardless of the parameters. We executed the experiments 100 times using these exact quantities of generations, and we observed that most experiments with larger *EnemyPool* sizes initially achieved better fitness, likely due to the greater variety of enemies positioned in the early generations. By the end of the experiment, both sizes converged to similar results. For experiments with $N_{rooms} = 5$ it is possible to notice an advantage in fitness when using the larger *EnemyPool*, resulting in 0.98 for size 20 and 0.91 for 10. In the experiment of $N_{rooms} = 10$ there is an advantage in using size 20, resulting in the better fitness of 0.95 compared to 0.92 achieved by a larger *EnemyPool*, this could indicate that the larger the number of rooms the less effective *EnemyPool* is in improving the overall fitness. The last experiment with $N_{rooms} = 20$ presents an insignificant difference for both sizes, obtaining the result 0.85 for the *EnemyPool* of size 40 and 0.83 for size 80. Finally, the final fitness is lower than in previous experiments because of the increased effort required to balance a more significant number of rooms, making the GA inefficient for managing large rooms.

5. CONCLUSIONS

This work introduced the development of three algorithms, which endeavor to generate levels procedurally, thereby replacing or at least alleviating the burden on game designers to create levels manually. Alongside the two genetic algorithms employed for dungeon and enemy generation, a third algorithm was devised to control NPCs within the



((a)) *EnemyPool*: 10 20 and 40

((b)) Twice the size for the *EnemyPool*: 20, 40 and 80

Figure 3. Results for creating enemies.

game, presenting an intriguing alternative to traditional AI methods. The two GAs, for room generation and enemies, were evaluated through experiments to analyze their expressiveness and efficiency and the impact of different parameters on their performance. For the room generation GA, tests with varying grid sizes and room numbers showed a rapid increase in room count, followed by optimization of the diameter and average degree. There was a higher penalty rate for tiny rooms and narrow corridors in scenarios with limited space and many rooms. The algorithm ultimately succeeded in optimizing all scenarios, with greater ease observed for smaller numbers of rooms. The GA for creating enemies also yielded good results. We tested different numbers of rooms and sizes for the EnemyPool. There was significant fitness maximization for smaller room sizes, whereas the algorithm did not converge to values greater than 0.90 (with a maximum of 1) for more substantial numbers of rooms. Initially, a larger EnemyPool size sped up fitness optimization. Still, different sizes produced similar results over time, indicating that this parameter only significantly benefits the search for a solution after the initial generations.

References

[Adams 2009] Adams, E. (2009). *Fundamentals of Game Design 2nd Edition*. New Riders, 2 edition.

[Brown et al. 2017] Brown, J. A., Lutfullin, B., e Oreshin, P. (2017). Procedural content generation of level layouts for hotline miami. In *2017 9th Computer Science and Electronic Engineering (CEEC)*.

[de Lima et al. 2019] de Lima, E. S., Feijó, B., e Furtado, A. L. (2019). Procedural generation of quests for games using genetic algorithms and automated planning. In *SBGames*, pages 144–153.

[Ruela e Guimarães 2014] Ruela, A. S. e Guimarães, F. G. (2014). Coevolutionary procedural generation of battle formations in massively multiplayer online strategy games. In *2014 Brazilian Symposium on Computer Games and Digital Entertainment*.

[Russel e Norvig 2013] Russel, S. e Norvig, P. (2013). *Inteligência Artificial 3^a.ed.* Elsevier Editora Ltda, Rio de Janeiro, Rio de Janeiro.

[TOGELIUS et al. 2011] TOGELIUS, J., YANNAKAKIS, G. N., STANLEY, K. O., e BROWNE, C. (2011). Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186.