# The Procedural Content Generation Benchmark: An Open-source Testbed for Generative Challenges in Games

Ahmed Khalifa
Institute of Digital Games
University of Malta
Msida, Malta
ahmed.khalifa@um.edu.mt

Roberto Gallotta
Institute of Digital Games
University of Malta
Msida, Malta
roberto.gallotta@um.edu.mt

Matthew Barthet
Institute of Digital Games
Institute of Digital Games, University of Malta
Msida, Malta
matthew.barthet@um.edu.mt

Antonios Liapis
Institute of Digital Games
University of Malta
Msida, Malta
antonios.liapis@um.edu.mt

Julian Togelius
Game Innovation Lab
New York University
Brooklyn, USA
julian@togelius.com

Georgios N. Yannakakis
Institute of Digital Games
Institute of Digital Games, University of Malta
Msida, Malta
georgios.yannakakis@um.edu.mt

## Abstract

This paper introduces the Procedural Content Generation Benchmark for evaluating generative algorithms on different game content creation tasks. The benchmark comes with 12 game-related problems with multiple variants on each problem. Problems vary from creating levels of different kinds to creating rule sets for simple arcade games. Each problem has its own content representation, control parameters, and evaluation metrics for quality, diversity, and controllability. This benchmark is intended as a first step towards a standardized way of comparing generative algorithms. We use the benchmark to score three baseline algorithms: a random generator, an evolution strategy, and a genetic algorithm. Results show that some problems are easier to solve than others, as well as the impact the chosen objective has on quality, diversity, and controllability of the generated artifacts.

## CCS Concepts

• **Applied computing** → **Computer games**; • **Theory of computation** → **Evolutionary algorithms**.

## Keywords

Procedural Content Generation, Search-Based Generation, Evolutionary Algorithms, Evaluation, Benchmark

**ACM Reference Format:**
Ahmed Khalifa, Roberto Gallotta, Matthew Barthet, Antonios Liapis, Julian Togelius, and Georgios N. Yannakakis. 2025. The Procedural Content Generation Benchmark: An Open-source Testbed for Generative Challenges in Games. In *International Conference on the Foundations of Digital Games (FDG '25), April 15–18, 2025, Graz, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3723498.3723794



*FDG '25, Graz, Austria*

ACM ISBN /25/04
https://doi.org/10.1145/3723498.3723794

## 1 Introduction

Scientific and technological progress requires reliable means of measuring the performance of methods and machines. Therefore, extensive efforts focus on developing tools and techniques for such measurements within various scientific and engineering fields. In Artificial Intelligence (AI), *benchmarks* are crucial to the field: most AI papers include a comparison with the state of the art on some benchmark. In Reinforcement Learning (RL) research, *Montezuma's Revenge* (Utopia Software, 1984) and *Pitfall!* (Activision, 1984) were two games—among 100 games in the Arcade Learning Environment [4]—which showcased the problems with deep RL approaches and pushed researchers to innovate [14]. However, some qualities are easier to benchmark than others. While an algorithm's speed and memory footprint are easily measurable, qualities related to creative expression are inherently complicated to measure—especially via automated means. This does not make such benchmarks less important [20].

While it may be hard to directly measure the creativity of the output of a software system [45], one can measure the usefulness of software that forms part of a creative system. Good measures of usefulness would assess how well the output performs according to some functionality measure, how diverse the output is, and how well the generator can be controlled. Having quantifiable measures of usefulness allows AI researchers to identify and address limitations in current algorithms. There are currently very few benchmarks in Procedural Content Generation (PCG) and generative AI in games (see Section 2.2). A one-stop framework that researchers and newcomers can use to explore the generative problems in games can lead to more rigorous testing protocols that can change the frontier of PCG research—in the same way that Arcade Learning Environment changed RL research.

In this paper, we introduce the Procedural Content Generation Benchmark (PCG Benchmark) which marks the first step towards standardizing problems in the generative space in games. It offers a set of problems where researchers can explore different methods and understand what works and what does not, as well as an extensible framework for adding further PCG problems. The

PCG Benchmark comes with 12 problems out of the box, including the generation of game rules, levels, buildings, word games, and patterns (see Section 4.1). Every problem has to follow the same evaluation criteria, using an array of content as input and returning the success score in terms of quality, diversity, and controllability as output (see Section 3). Reaching the maximum score on these problems does not necessitate that challenges in generating content for this game have all been overcome. It just means that this particular formulation of the problem with these particular criteria is solved. We test the framework against three baseline algorithms that follow the search-based PCG paradigm [53]; results showcase the different challenges posed by the multi-faceted generative problems already implemented into the PCG Benchmark.

## 2 Background

This section takes a brief look at the history of PCG in games and the benchmarks that have advanced Game AI research.

### 2.1 Procedural Content Generation in Games

PCG is central to technical games research. Early applications of PCG in games focused on adding scale and replayability to games with limited hardware, such as the infinite level layouts of *Rogue* (Toy and Wichman, 1980), or the large star systems provided in *Elite* (Acornsoft, 1984). In *search-based* PCG [53], artificial evolution or similar stochastic optimization methods are used to generate content that optimizes an evaluation function. Search-based PCG provides the capacity to generate more complex content while retaining functionality guarantees. Some notable examples of search-based PCG include generating weapons for *Galactic Arms Race* [22], generating new level layouts for *Super Mario Bros.* (Nintendo, 1985) [48], and generating novel suggestions as a designer assistant tool [10, 38]. Experience-driven PCG [58] combines the search-based approach with player models to generate content that elicits a desired experience in the player, such as *Super Mario Bros* levels [57] which maximize a diversity metric based on Raph Koster's *theory of fun* [33], or levels which elicit emotional trajectories during play [39].

Within search-based PCG, quality diversity (QD) evolutionary algorithms [44] can be used to generate a set of high-quality solutions across a range of behavior metrics to ensure meaningful diversity in the output [17]. PCG through QD has become a popular method for generators focused on creativity, such as for generating novel game levels [5], bullet patterns [29], interesting 2D [34, 37] and 3D [15] spaceships, and diverse *Minecraft* (Mojang, 2011) buildings [3].

Perhaps the biggest downside of search-based PCG is the computational cost of content production. PCG via machine learning (PCGML) [51] instead moves the computational cost to the training phase, with generally much faster inference. Many examples of PCGML in games leverage self-supervised learning on existing game content to learn to generate game levels, e.g. using generative adversarial networks [56], wave function collapse [32], or computer vision [21]. But this requires a sufficient amount of game content to train on. PCG via reinforcement learning (PCGRL) [27] instead trains RL agents to generate new content based only on rewards, and can generate new content in an online fashion [57].

Looking at the evolution of PCG over time [35], we are always pushing towards using new and novel systems to generate content. Establishing a standardized and open-sourced approach for evaluating the capabilities of these algorithms ensures transparency and replicability of results [6]. With the rise of new AI methods [16], having a comprehensive way to compare large language models or other generative AI methods to earlier algorithms is crucial. Importantly, an easy-to-use benchmark can act as a teaching tool for PCG at an undergraduate or graduate level, allowing students to compare their algorithms' performance.

### 2.2 Game AI Benchmarks

AI has a long-standing history of using game-based benchmarks as a point of comparison between new methods. *Chess* and *Go* famously played a significant role in the development of AI with the development of *DeepBlue* [9] and *AlphaGo* [49] respectively. The field eventually transitioned to tackle the complexity of digital games such as the *Mario AI* benchmark [25] and *Starcraft II* (Blizzard, 2011) [55]. Training gameplaying agents via RL [40] has relied on standardized and readily available benchmarks built on *OpenAI Gym* [7] as an easy point of comparison between methods. A notable example is the *Arcade Learning Environment* [4], which consists of over 100 game environments for the *Atari 2600* console spanning a variety of game genres. The development of these benchmarks was pivotal towards the advancement of gameplaying AI [40]. Newer gameplaying benchmarks even include a player experience component [2], targeting more believable and human-like play.

With the rise of generative AI, similar benchmarks are needed to push toward better methods that can work for games. The history of PCG research already includes various PCG testbeds and competitions, such as competitions on *Super Mario Bros.* (Nintendo, 1985) [23], Generative Design in *Minecraft* (GDMC) [18, 46], and the level and rule generation tracks in GVGAI [43]. However, there are currently no unified benchmarks for PCG which encompass a variety of problems and facets of content generation. One critical difference between the benchmarks for PCG and gameplaying is that evaluating the output of creative systems is often ill-defined, complicated to measure, and subjective. For example, the GDMC [46] competition employs human judges to evaluate the submitted algorithms. This human-centered evaluation cannot scale well and can be influenced by biases [59]. Evaluating the output of generated content is usually problem-specific due to the huge variety in content types and representations. Through our benchmark, we aim to provide a standardized platform for an easier comparison of generators across a common set of problems.

## 3 PCG Benchmark

The PCG Benchmark[1] is an easy-to-use framework that allows users to evaluate their generative algorithm against a multitude of generative problems in games. The framework follows the design concepts of *OpenAI Gym* [7]: each problem is independent and has its own representation and evaluation criteria. The framework provides an evaluation function that can be used to evaluate any content on three criteria:

[1] https://github.com/amidos2006/pcg_benchmark

- **Quality** measures the percentage of the input content that passes the quality criteria for the current problem. For example, if the problem is generating levels for Mario with quality criteria of having playable levels and the user provided 100 levels to evaluate, the system will test all the levels, and if 20 levels are playable then the result will be 20%.
- **Diversity** measures the percentage of the input content that passes the diversity criteria for the current problem. This is an important aspect in evaluating any PCG algorithm as having a generator that generates the same content or small variants of the same content every time should be considered a bad generator [11]. For example, if the problem is generating full games, having a generator providing 100 games where all of them are variants of a block-pushing game (a-la *Sokoban*) with just different named objects then diversity should be low.
- **Controllability** measures the percentage of the input content that adheres to some controllability constraints. Controllability has to be tested against a control parameter that the user should provide; such parameters are designer choices (e.g. the number of enemies in a generated level), rather than a playability constraint. Without having control parameters paired with the input content, controllability can not be measured; the system will always return 0%. Not every generator needs to be controllable: developers can always create a new generator for every different problem. However, having a generator that adapts to target parameters without changing any of the underlying code makes it easier to use during production.

Besides these percentages, the benchmark returns additional details about every artifact in the input. Each artifact receives three values within $[0, 1]$ that capture how close it is to passing each criterion (quality, diversity, or controllability). This can be used as an error function for PCGML, as a reward for PCGRL, or as a fitness function for search-based PCG (see Section 2.1). Additional, problem-specific results can also be provided per artifact. For example, in the *Zelda* problem (see Section 4.1), the solution length and level connectivity are part of the additional information that is returned per generated level in the input. This additional information can help the user to create more complex generators such as Quality Diversity algorithms [17], using these metrics as behavior characterizations [41] for the generated content.

The framework also provides two possibility spaces [12], one for the content and one for the control parameters. These spaces are designed to be similar to the *OpenAI Gym* spaces for observations and actions [7]. Our spaces, however, define the possible values for both content and control parameters. These two spaces can be used to sample randomly from, validate if a content or a control parameter is possible, mix two artifacts probabilistically, change parts of an artifact randomly, convert an artifact to a flat (string) representation and back, etc. These functionalities allow users to build their generators with less friction. For example, a random generator just samples randomly from the content space and can keep the best-discovered artifacts. Moreover, access to these spaces allows the framework to have different representations for each generative problem: the user must then build their generator to work with each representation. This allows the system to work with any type of content in games such as text, level generation, rule generation, patterns, etc. Besides these functions, the framework has a render function that converts the content into the corresponding graphical representation. This graphical representation depends on the problem at hand: images, videos, strings, sounds, etc.

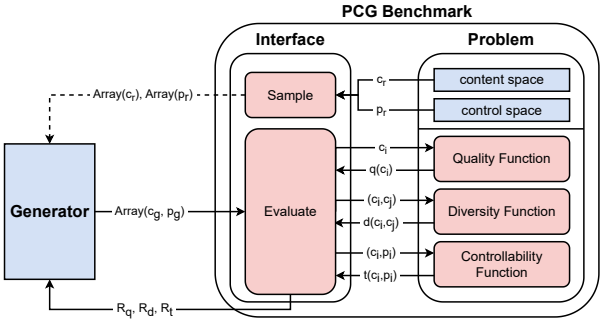


**Figure 1: The system diagram for the PCG Benchmark that showcases how to use the framework. First, the Generator can sample an array of random content ($Array(c_r)$) and an array of random control parameters ($Array(p_r)$). Then the Generator returns an array of paired content and control parameters ($Array(c_g, p_g)$) to be evaluated. The system sends them to the current problem to calculate their values ($q(c_i)$ is the quality value for a content, $d(c_i, c_j)$ is the diversity value between two content, and $t(c_i, p_i)$ is the controllability value between a content and a control parameter). Finally, the system returns the results for quality ($R_q$), diversity ($R_d$), and controllability ($R_t$).**

## 3.1 Using the Benchmark

Figure 1 shows the system diagram of the PCG Benchmark and how a **Generator** can interact with all the moving parts through a simple interface. The user needs first to specify the problem that they need to solve (see section 4.1). After that, the user can build their generator using any method, including constructive [47], search-based [53], quality diversity [17], machine learning [51], RL-based [27], constrained-based [26], and other methods. The generator can use the control space, content space, and the evaluate function as many times as needed, even integrating them in its internal generative loop, e.g. in search-based methods [53]. Finally, the generator will return the generated content paired with the control parameters that this content was generated for[2], and the benchmark will evaluate the content and return the scores for quality, diversity, and controllability (if applicable).

## 3.2 Extending the Benchmark

To make the framework easy to extend, we separated the interface where the generator interacts from the actual generative problem. When a user wants to test their algorithm, they ask the system for a specific problem using its unique name. The system finds it from

[2]If values for the controllability parameters are not provided by the user, the algorithm returns the generated content only.

the list of all the registered problems and returns it inside the fixed environment interface. Figure 1 shows a constructed environment for a specific problem. Due to this separation, adding new problems is easy. The user needs to create a new problem class that has the following functions:

- **Info Function** takes as input the artifact and outputs an object that contains all the needed and related information of that content. This function is crucial, as it is used by most other functions (below). For example, if we are evaluating mazes, we might need to have the maze solution precomputed for quality (have a maze with at least 20 steps to solve), diversity (mazes with different solutions), and controllability (have a maze with exactly $X$ steps to solve where $X$ is the control parameter).
- **Quality Function** takes the info object and returns a value within $[0, 1]$ that reflects how close that content is to passing the quality criteria (at 1.0 it passes). In our maze generation example, this could be how close the solution length is to 20 or more.
- **Diversity Function** takes as input two info objects for different artifacts and outputs a value within $[0, 1]$ that reflects how similar these two artifacts are to each other (1.0 means they are different and 0.0 means they are identical). In our maze generation example, diversity can be on the actions taken for solving the maze, and if the solutions are at least 5 actions different between the two mazes then diversity is 1.0.
- **Controllability Function** takes as input the info object and value(s) for its control parameter(s) and outputs a value within $[0, 1]$ that reflects how close the content is to match these values. If a control parameter is the number of enemies in a generated level, the function will return 1.0 if the generated has a number of enemies within an allowed range of this parameter's values.
- **Render Function** takes as input the artifact and outputs its final representation. This final representation could be a graphical representation or any other format as needed: image, sound, video, string, etc. In our maze generation example, the function will return a black-and-white image where white pixels are passable tiles and black pixels are solid tiles.

Besides these functions, the content space and control space have to be defined, since each problem can have its own complex representation and control parameter. For example, the *Arcade Rules* problem (see Section 4.1) has a representation that explains the rules of the game and game object locations and a control parameter which is a 2D layout of solid and empty tiles which constitutes the level that the rules should work on.

# 4 Experiments

This section covers the experimental protocol used to test our benchmark[3]. In Section 4.1, we describe each of the generative problems presented in the benchmark. Section 4.2 focuses on describing the baseline generators used to test them. Finally, Section 4.3 explains the representation used and the different fitness functions used during our experiments.

[3]Available at https://github.com/amidos2006/benchmark_experiments

## 4.1 Problems

The PCG Benchmark includes 12 PCG problems, each with its own representation, control parameters, quality, diversity, and controllability criteria. We will summarize all of these problems here but more details are found in the documentation[4].

**Arcade Rules** hinges on generating the rules for new 2D arcade games based on the framework provided in [52]. The problem consists of a dictionary of integers (including the starting coordinates, winning conditions, movement rules, and collision rules) mapped to different effects. For example, if the win condition is 0, the game is won if the player is alive after 40 frames since the game starts. Seven quality criteria ensure that the game can be won, lost, and is playable by several different agents (e.g. static or random) that reach different performance profiles. The control parameter is a $7 \times 7$ binary array of the level layout (each tile is solid or empty); if not provided, a simple fixed layout is used.

**Binary** originates from the PCGRL framework [27], and involves generating fully connected 2D mazes, consisting of empty or solid tiles. The default variant of the problem is to generate $14 \times 14$ mazes, and the quality constraint is having a minimum length of the longest path between any two empty tiles in the maze of at least 28 tiles. The problem has one control parameter, further specifying the minimum length of the longest path.

**Building** is inspired by [24], which presents the challenge of generating buildings in a 3D space with four different types of Lego blocks ($1 \times 1$, $1 \times 3$, $3 \times 1$, and $3 \times 3$ voxels). The default variant of the problem cares about generating buildings that use 40 Lego block types, must have a maximum size of $7 \times 7 \times 12$ voxels, and must be taller than 6 voxels (quality constraints). Four control parameters determine the ratio of different blocks to be used (out of 40).

**Dangerous Dave** hinges on generating level layouts for a small discrete version of the DOS game *Dangerous Dave* (Uptime Disk Monthly, 1988). The game is a small platformer where the player must avoid spikes, collect diamonds, and reach the exit. Generated levels have multiple quality constraints (number of tiles per type, solvability by AI agents, minimum number of jumps, and reachability of all diamonds). The default variant of the problem has $11 \times 7$ tiles and the solution must have at least 2 jumps. The problem has five control parameters (coordinates of start tile and exit tile, and number of diamonds required).

**Elimination** uses the word game *Elimination* (Khalifa, 2018) described in [28]. The problem is to generate a sequence of letters that can create at least one short word, one long word, and nothing longer. The problem has five quality constraints (including words allowed). The default variant of this problem presents sequences of 8 letters and requires the short words to lie between 40% and 60% of the most common English words and the long words to lie between 60% to 80% of the most common English words. The problem has one control parameter, determining the maximum consecutive letters of an actual word in the initial sequence.

**Isaac** hinges on generating fully connected dungeons for a simplified version of *The Binding of Isaac* (McMillen, 2011) video game. The dungeons generated must contain a target number of rooms, including a starting room, a boss room, a treasure room, and a shop room. Five quality constraints assess whether the dungeon

[4]Available at https://github.com/amidos2006/pcg_benchmark

is connected, whether all special rooms are present, the minimum distance between certain rooms, etc. The problem has one control parameter (the number of rooms in the dungeon).

**Lode Runner** is based on a simplified version of the *Lode Runner* (Broderbund, 1983) puzzle platformer video game. The goal is to generate a playable level with a minimum number of gold and enemies. Generated levels must fulfill seven quality constraints, including minimum tiles per type, and having most tiles reachable to the player. The problem has two control parameters (the number of ladder tiles and rope tiles). The default variant of this problem requires levels of size $32 \times 22$ tiles, encoded as a grid of $16 \times 11$ indices (with $2 \times 2$ tiles per index), with a minimum of 6 gold items and 3 enemies.

**MiniDungeons** is based on the *MiniDungeons* framework [36], where the aim of the game is to reach the exit without dying to enemies. This variant uses deterministic combat when facing enemies. Four quality constraints define the minimum number of tiles per type in the level, whether the dungeon is connected, and whether at least some monsters are defeated along the shortest path to the exit. The default variant of the problem is to generate a solvable $8 \times 12$ level which forces the player to kill 12 enemies before reaching the exit. The problem has five control parameters (coordinates of start tile and exit tile, target number of treasures).

**Super Mario Bros** is inspired by the work done in [13] where *Super Mario Bros.* levels are represented as a sequence of vertical slices sampled from the original game. The default variant of the problem is to generate a playable level made of 150 slices with a similar look to original Mario levels. Five quality constraints assess whether the level can be completed by an A* agent, has flat areas or small elevations, unbroken pipe structures, and few floating enemies. The problem has three control parameters (number of enemies, coins, and jumps achieved during an AI playthrough).

**Sokoban** is based on the Japanese block-pushing game (Imabayashi, 1982) and a generator must create fully solvable puzzles. The default variant of this problem requires generating playable levels of $5 \times 5$ tiles; five quality constraints test the presence of special tiles (player, crate) and that an A* agent can solve the level in at least 10 moves. The problem has one control parameter (target number of crates).

**Talakat** generates bullet patterns for the Talakat shoot-em-up game [30]. Four quality constraints assess the quality of the bullet patterns (e.g. maximum bullet spawners, minimum bullets per frame, distribution of bullets in different parts of the screen, etc.). The problem's control parameter is a 1D array of the distribution of bullets over time within one second of gameplay.

**Zelda** originates from the General Video Gameplaying AI framework [43], and has been used in several research papers [27, 50, 54]. The generated level is a maze with a key, a door, and enemies: the player must find a key to get to the exit without dying. Six quality constraints test the connectivity of the level, the number of tiles of each type, and the minimum length for the solution. The problem has two control parameters (the minimum distance from the start to the key tile and the minimum distance from the key to the door tile).

## 4.2 Generators

In this initial study, we test three baseline generators on the PCG Benchmark. All generators follow simple *search-based* PCG approaches [53]. Search-based algorithms can be used for black-box optimization; they do not need to understand (or adapt to) the problem [1]. The generators were tested on the default variants for all of the described problems, and each experiment was run in 10 independent runs. All methods were required to produce 100 individuals during each generation. All the generators ran for 200 generations. The implementation of each generator is as follows:

- **Random Generator (Random):** Every generation, a new population of individuals is randomly created and evaluated. The new population is combined with the previous generation, and the best 100 individuals are kept for this generation.
- $\mu + \lambda$ **Evolutionary Strategy (ES):** Every generation, a new population of 100 ($\lambda$) is mutated from the previous generation and evaluated, with the best 100 ($\mu$) individuals kept between generations. Candidates are evolved using a uniform mutation rate of 5%.
- **Genetic Algorithm (GA):** This method extends ES by introducing selection and crossover operators. We use tournament selection on 7 individuals to select candidates that produce new offspring. Uniform crossover is used to combine parents, with a crossover rate of 50%. Offspring have a 5% mutation rate, same as ES. The population size is 100 and elitism preserves the best 10 solutions between generations.

## 4.3 Representation and Fitness Functions

The chromosome for all three algorithms consists of two parts, a content vector and a control parameter vector. These are sampled at the beginning randomly from the content space and control space (explained in Section 3). For the sake of simplicity, operators only work on the content part; nothing changes the control parameter (however, different solutions satisfy different controllability constraints). Looking into the fitness function, we use 3 different fitness functions that care about quality, controllability, and diversity:

- **Quality Fitness (Q):** Solution fitness is equal to the quality of the artifact. This fitness does not check if controllability constraints are met. The formula is shown in Eq. (1).

$$f(c_i, p_i, \mathbb{C}) = q(c_i) \tag{1}$$

  where $c_i$ is the content being evaluated, $p_i$ is the control parameter associated, $\mathbb{C}$ is the total population of content, and $q(c_i)$ is the quality value for input content.
- **Quality then Controllability Fitness (QT):** This fitness function first tries to maximize quality, then controllability if the solution is optimal with regard to quality. The fitness is shown in Eq. (2).

$$f(c_i, p_i, C) = \begin{cases} \frac{1}{2} q(c_i) & \text{if } q(c_i) < 1 \\ \frac{1}{2}(q(c_i) + t(c_i, p_i)) & \text{if } q(c_i) = 1 \end{cases} \tag{2}$$

  where $c_i$ is the content being evaluated, $p_i$ is the control parameter associated, $C$ is the total population of content, $q(c_i)$ is the quality value for input content, and $t(c_i, p_i)$ is

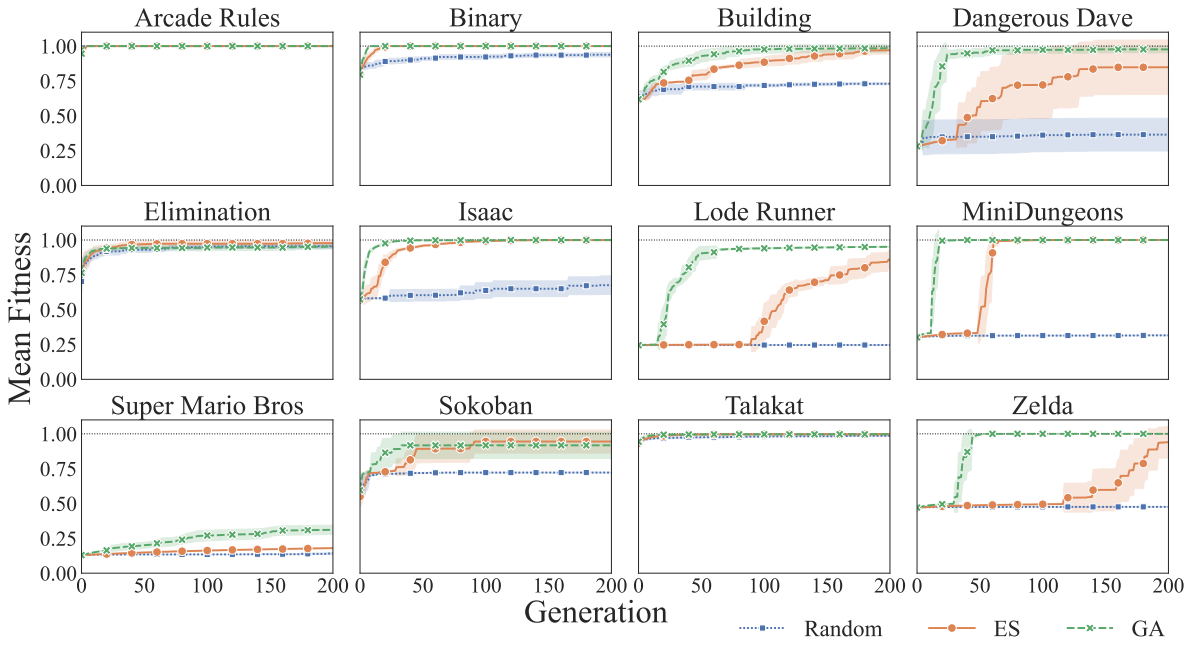


**Figure 2: Progression of the maximum fitness when optimizing the Quality fitness with the three baseline algorithms. Results are averaged from 10 runs, with 95% confidence intervals as the shaded area.**

the controllability value for the input content with respect to the input control parameter.

- **Quality, Controllability, then Population Diversity Fitness (QTD):** Similar to the previous function, this fitness function first tries to maximize quality, then after that controllability if the solution is optimal with regard to quality. Finally, if the solution is optimal with respect to quality and controllability, it tries to optimize towards population diversity as shown in Eq. (3). Population diversity is challenging as it fluctuates depending on the current population, but it can help search algorithms overcome local optima.

$$f(c_i, p_i, C) = \begin{cases} \frac{1}{3} q(c_i) & \text{if} \quad q(c_i) < 1, \\ & \quad t(c_i, p_i) < 1 \\ \frac{1}{3} (q(c_i) + t(c_i, p_i)) & \text{if} \quad q(c_i) = 1, \\ & \quad t(c_i, p_i) < 1 \\ \frac{1}{3} (q(c_i) + t(c_i, p_i) + d(c_i, \mathbb{C})) & \text{if} \quad q(c_i) = 1, \\ & \quad t(c_i, p_i) = 1 \end{cases} \tag{3}$$

where $c_i$ is the content being evaluated, $p_i$ is the control parameter associated, $C$ is the total population of content, $q(c_i)$ is the quality value for input content, $t(c_i, p_i)$ is the controllability value for the input content with respect to the input control parameter, and $d(c_i, \mathbb{C})$ is a measure of uniqueness of $c_i$ with respect to the set of content $\mathbb{C}$ in the population.

## 5 Results

In our experiment, we test three search-based algorithms (see Section 4.2) using three different fitness functions to optimize content for different problems. Each problem has its own measures for what constitutes good (i.e. *feasible* content $c_i$ if $q(c_i) = 1$) and diverse (i.e. *unique* content if its $d(c_i, \mathbb{C}) = 1$ with respect to a set of content $\mathbb{C}$). Moreover, in the initial population, each individual has its own control parameters randomized within allowed value ranges. While the control parameters are not randomized, mutated, or recombined *during* evolution, the search process will prioritize copying over (via elitism, $\mu$+$\lambda$, or choosing the best 100 individuals in Random) solutions with more easily satisfied control parameters. Each problem similarly has its own controllability score, based on the content $c_i$ and its paired control parameters $p_i$: controllability criteria are satisfied if $t(c_i, p_i) = 1$; as shorthand, we label such individuals *controlled* in the below section.

Figure 2 shows how the maximum fitness (when optimizing for Quality alone) improves over the generations for the different algorithms. We observe that some problems can discover feasible solutions even with random initialization: *Arcade Rules* and *Talakat* have near-optimal individuals in the initial population, which leads to almost no improvement from evolution. Some problems struggle to discover high-quality individuals with random initialization: *Lode Runner*, *MiniDungeons* and *Dangerous Dave* start with a maximum fitness around 0.25 before evolution, and *Super Mario Bros* start from around 0.13. Evolution improves the maximum fitness in all problems, although to different degrees depending on the problem. GA has overall slightly better improvements in maximum fitness from initial to final population (between 6% relative increase in maximum fitness for *Talakat* and 288% relative increase for *Lode*

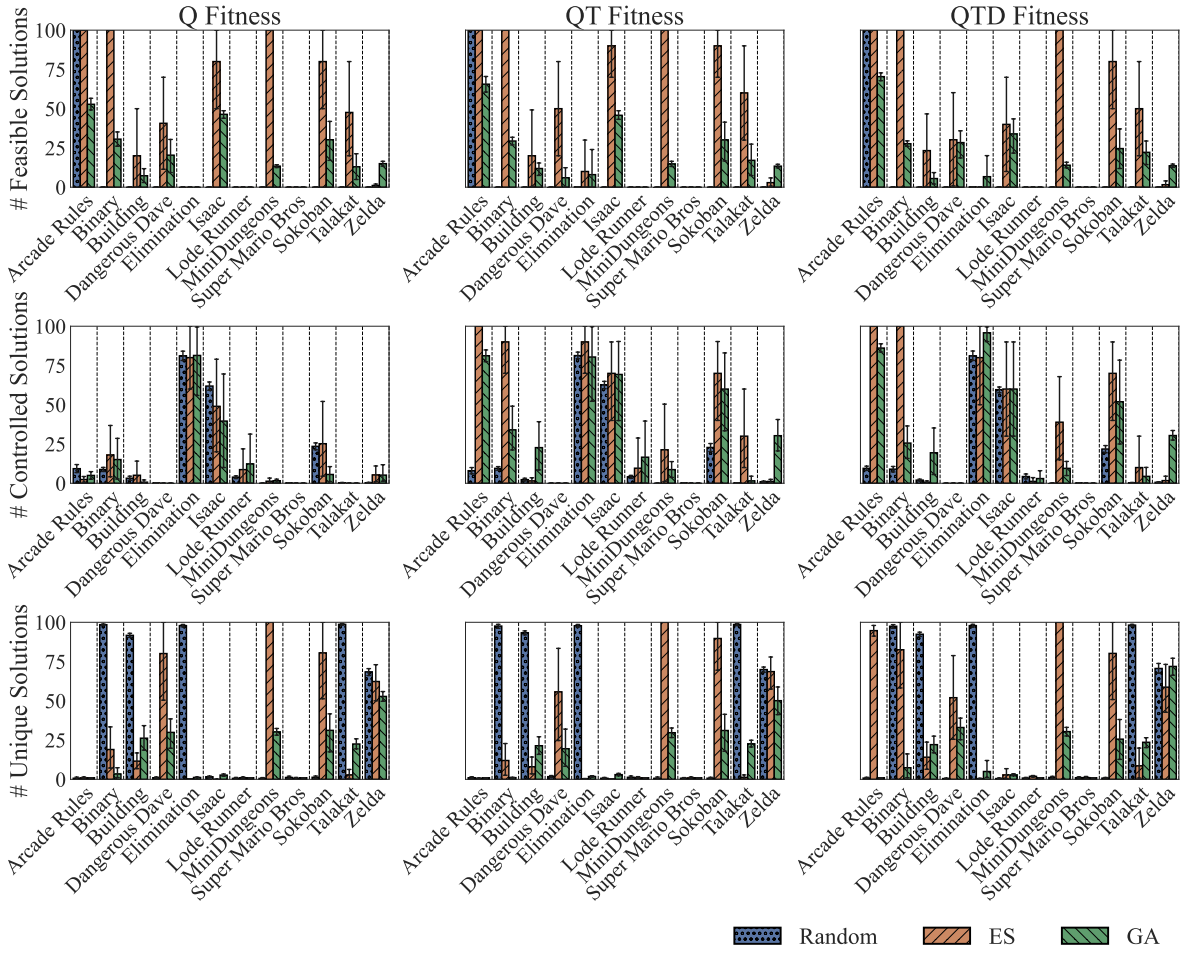


**Figure 3: Number of solutions $c_i$ (out of 100) in the final population (after 200 generations) that are feasible ($q(c_i) = 1$), controlled ($t(c_i, p_i) = 1$), and unique ($d(c_i, \mathbb{C}) = 1$ compared to the final population $\mathbb{C}$). Results are averaged from 10 runs, with 95% confidence intervals as error bars.**

*Runner*), with ES a close second (between 5% relative increase in *Arcade Rules* and 251% relative increase for *Lode Runner*). These relative increases can be noticed in the generated examples in tables 1 and 2 where the best random content looks subjectively more noisy compared to its counterpart (especially in *Lode Runner*, *Super Mario*, and *Zelda*). Random does not increase maximum fitness as much, with the highest relative increase of 36% from the initial population in *Elimination* and negligible increases (below 1%) in *Lode Runner* and *Zelda*. In *Super Mario Bros*, however, no solution after 200 generations satisfies the quality constraints for any of the generators in any of the runs. In this problem, changes in the maximum fitness are overall slow (141%, 39%, and 9% relative increase from the initial population for GA, ES, and Random respectively). This indicates that either the variation operators implemented or the quality constraints pose challenges to any stochastic search approach. Given similar difficulties in generating other large platformer levels in *Lode Runner*, the size of the level (genotype) for *Super Mario Bros* could also be an issue.

While so far we focused on how easy it is to generate feasible content with random initialization and to improve on them via search-based PCG, we are ultimately most interested in the artifacts at the end of the process (after 200 generations). Figure 3 shows the number of feasible individuals ($q(c_i) = 1$) in the final population, the number of unique individuals ($d(c_i, \mathbb{C}) = 1$ where $\mathbb{C}$ is the final population in the same run), and the number of final individuals that satisfy all controllability constraints for their paired control parameters ($p_i$), i.e. $t(c_i, p_i) = 1$.

Observing the number of individuals that satisfy all quality constraints in the top row of Fig. 3, conclusions mirror those from the maximum Q fitness progression (Fig. 2). Some games are easy to find feasible individuals for: *Arcade Rules*, *Binary*, *MiniDungeons* end up with an entire population of feasible individuals in all runs of ES regardless of fitness being optimized. Interestingly, in *Arcade Rules* Random also reaches a final population full of feasible individuals, but no feasible individuals in the other three games. Other games are too hard to find feasible individuals for any algorithm or

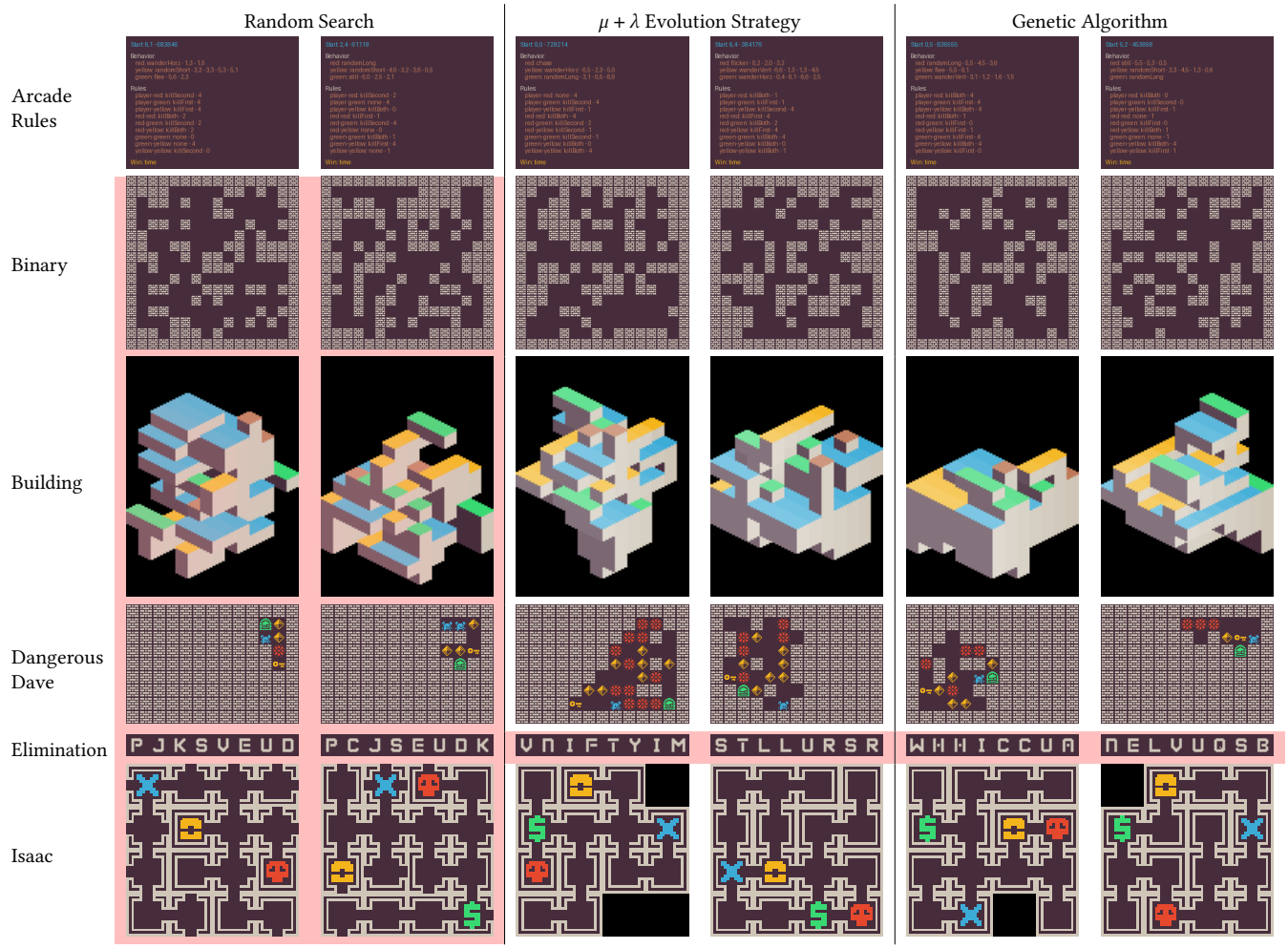


**Table 1: Example of the best generated content (one per run) for 3 different generators optimizing the *Quality* fitness function for the first 6 problems (see Section 4.1). Content with a red background means it failed the quality constraints.**

fitness target: *Super Mario Bros* and *Lode Runner* consistently have no feasible individuals in the final population. Interestingly, the choice of fitness target also impacts the number of feasible individuals: In *Isaac* (and in *Dangerous Dave* with ES) the number of feasible individuals dropped for QTD with respect to other fitness functions. Since the drop is not that big, it might be due to randomness. A special case is *Elimination*, where optimizing Quality alone leads to no feasible individuals while optimizing either QT or QTD leads to feasible individuals (for ES and GA with QT, and for GA with QTD). Given that both QT and QTD fitnesses would have values of 0 for infeasible individuals, this difference in behavior is an artifact of randomization (either in initialization or stochastic search process): indeed, feasible individuals were found only in one run of each algorithm and in one runs of GA for QTD fitness. Although *Zelda* has a small number of feasible solutions in the final population, the GA manages to find at least one feasible chromosome in 8 of 10 runs while ES does so in 8 runs for Q fitness, 6 runs for QT fitness and 5 runs for QTD fitness. GA in general does not contain many feasible individuals in the final population but is much more consistent in discovering feasible individuals than ES. Of a total of 360 runs across all environments and all fitness targets, the GA has feasible solutions in the final population in 224 runs, versus 191 runs for ES and 32 runs for Random. The fewer feasible solutions per population in the GA are likely due to the different elitism mechanisms: ES saves the best 100 solutions while GA saves the best 10 solutions.

Observing the number of individuals that satisfy all (paired) controllability constraints in the second row of Fig. 3, it is evident that some benchmarks are very difficult to satisfy controllability criteria. *Super Mario Bros* and *Dangerous Dave* rarely have controlled individuals for any algorithm and for any fitness target. *Lode Runner* is interesting because a few individuals are always controlled in every run of Random (regardless of fitness target), while GA and ES have few runs where any controlled individuals end up in the final population even with QT fitness (1 run for ES, 3 runs for GA) and with QTD fitness (1 run for ES, 2 runs for GA). In most

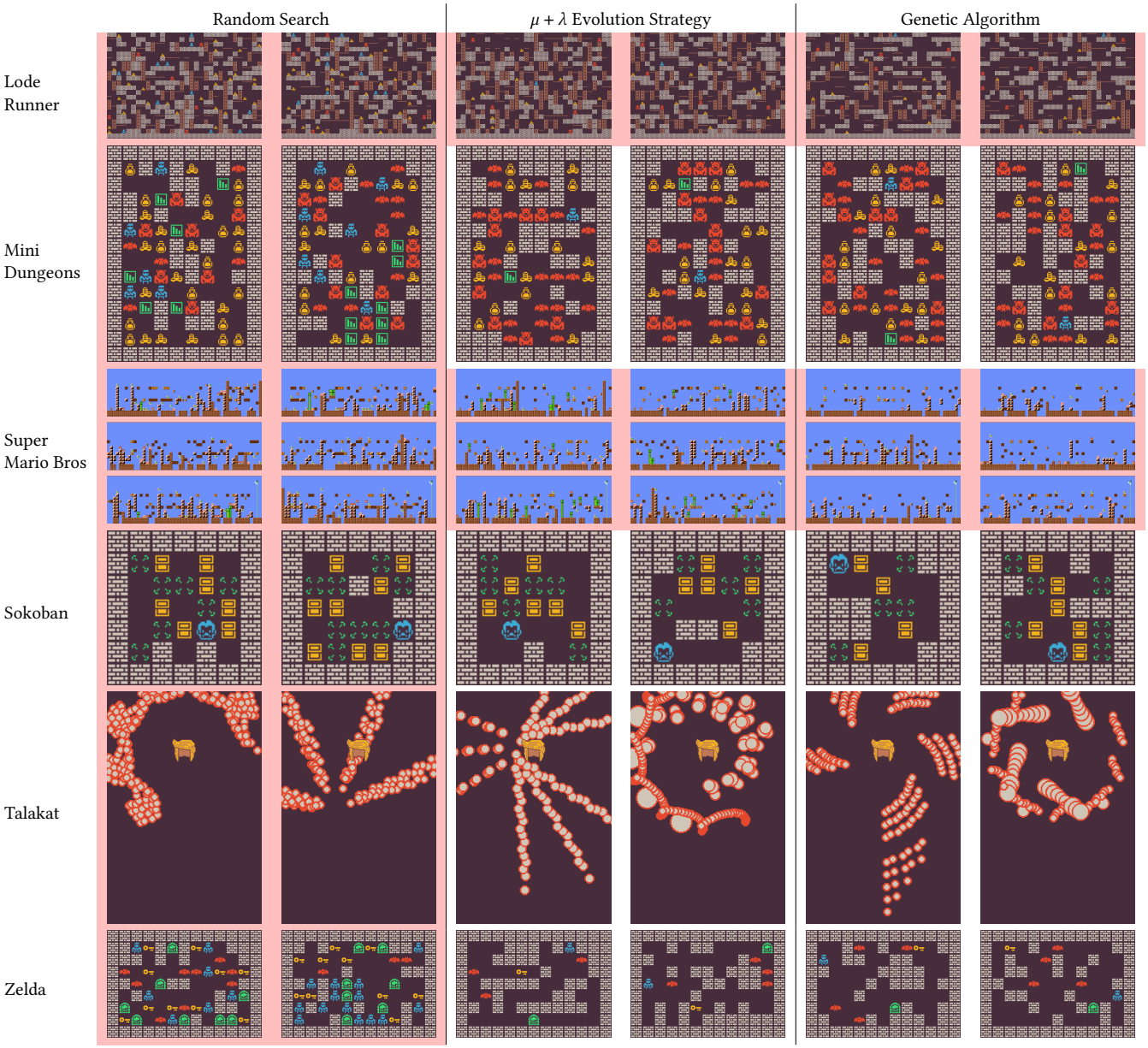


**Table 2: Example of the best generated content (one per run) for 3 different generators optimizing the *Quality* fitness function for the last 6 problems (see Section 4.1). Content with a red background means it failed the quality constraints.**

problems, applying pressure for controllability (via QT or QTD fitness) does have an effect, with higher numbers of controllable individuals than for Quality alone. An interesting finding is that for some benchmarks the QTD fitness (which has more objectives and is thus more diluted) leads to many more controlled individuals than QT fitness alone, especially *Talakat* for GA and *MiniDungeons* for ES. Worth noting is that, at least with QT and QTD fitness targets, some benchmarks are easy to find controlled individuals for (particularly *Arcade Rules*, *Binary*, *Elimination*, *Isaac*, and *Sokoban*).

From the number of unique individuals in the final population of each run (third row of Fig. 3), we observe that some benchmarks are easy to find unique individuals for (even if not explicitly targeting that as an objective via QTD fitness). *Binary*, *Building*, *Elimination*, and *Talakat* achieve many unique individuals with Random, fewer unique individuals with GA, and few if any with ES: this holds both for Q and QT fitness targets, which do not target diversity as an objective. For QTD, Random outperforms the other stochastic search methods in *Building*, *Elimination*, and *Talakat*. However, QTD pushes GA to maintain at least one unique individual in all

runs of all problems (even if for *Arcade Rules*, *Lode Runner* and *Super Mario Bros* it's always only one unique individual). Finally, *Super Mario Bros*, *Isaac* and *Lode Runner* struggle to maintain many unique individuals with any algorithm even when targeting QTD; however, it is important to note that *Isaac* manages to maintain many feasible and controlled individuals with this fitness target.

The above experiments highlight how different the problems included in the PCG Benchmark are in terms of generating feasible, unique, or controlled individuals. Results also indicate that different generators have different trade-offs in different problems: maximum fitness improves faster with more pressure towards convergence from the GA (see Fig. 2) while introducing random individuals (rather than modifying the population) leads to more unique final solutions at the cost of feasibility and controllability. While the goal of this experiment was not necessarily to compare algorithms, the $\mu + \lambda$ ES seems more suitable as a general approach with more feasible solutions, more unique solutions, and more controlled solutions overall. It is worth noting however that the GA is more reliable: considering all three fitness targets, the GA produces feasible solutions after 200 generations in 224 of 360 runs (10 runs, 12 problems, 3 fitness targets), compared to 191 runs for ES and 32 runs for Random. The extensibility of the current problems, however, opens more possibilities for follow-up experiments: indicative modifications could be applied to some harder problems (e.g. *Super Mario Bros* and *Lode Runner*) towards fewer quality constraints or smaller genotypes, or to hard-code controllability parameters rather than pair them with the solution as done in this paper.

## 6 Conclusion

This paper introduced a new benchmark to test any generative algorithm. The PCG Benchmark provides an interface that is easy to use and extend, similar to the *OpenAI Gym* [7]. The framework comes with 12 problems from the get-go that span across multiple different domains (rules, levels, structures, words, patterns, etc). We tested three different baseline algorithms (random, evolutionary strategy, and genetic algorithm) against all 12 problems, guided by different fitness functions that incorporate quality, controllability, and population diversity criteria. We noticed that generating large levels (i.e. *Super Mario Bros* and *Lode Runner*) was challenging for all baseline algorithms. Complex landscapes as in the *Elimination* problem were also not easy to solve due to low locality [53]. Interestingly, combining quality measures with controllability and population diversity sometimes helped the algorithm find better solutions faster.

Overall, we believe that the PCG Benchmark is a first step towards comparing generative algorithms. We expect that it will assist PCG research and education, in a similar that the *OpenAI Gym* [7] pushed RL research forward. We also believe that the benchmark is a great learning tool for newcomers to the field, as well as students. We note that solving a problem in the PCG Benchmark does not mean that this generative problem is solved for good or that the generated levels can be used for any type of human player; it means that the generator is good for a specific hypothetical player that the evaluation functions were designed for. For example, solving the *Super Mario Bros* problem with its default parameters (see Section 4.1) does not mean that you have the best generator for Mario levels. Instead, it means you have a generator that can create playable Mario levels with 15 non-floating enemies that follow the same tile distribution as the original Mario levels. Even playability for Mario is not human but proxied using A* algorithms [25]. We also want to note that the benchmark does not try to solve generality in the PCG domain. This means that the framework is not designed to have one agent that can tackle all problems but is more similar to *OpenAI Gym* [7] where each problem has its own representation and can be solved. We decided not to tackle the generality problem for the sake of simplicity, usability, and learnability; there are other frameworks that tackle this problem [31]. The problems in the PCG Benchmark act as milestones for researchers and students; once reached, users can expand the PCG Benchmark with new problems or harder variants of current problems (by adjusting their parameters in Section 4.1). In the end, having a standardized way to compare generative algorithms will shed light on the strengths and the drawbacks of PCG algorithms, pushing towards novel solutions as exhibited in gameplaying AI [14] to address hard exploration problems within the Arcade Learning Environment [4].

## Acknowledgments

Thanks to Kenny for creating 1-Bit Pack[5] which was used for most of the 12 problems in the benchmark. Even the ones that did not use it were inspired by the color palette used in that pack.

This project has received funding from the Malta Council for Science and Technology (MCST) through the SINO-MALTA Fund 2022, Project OPtiMaL.

## References

[1] Charles Audet. 2022. Blackbox optimization. In *Encyclopedia of Optimization*. Springer.
[2] Matthew Barthet, Roberto Gallotta, Ahmed Khalifa, Antonios Liapis, and Georgios N. Yannakakis. 2024. Affectively Framework: Towards Human-like Affect-Based Agents. In *Proceedings of the International Conference on Affective Computing and Intelligent Interaction Workshops and Demos*. IEEE.
[3] Matthew Barthet, Antonios Liapis, and Georgios N Yannakakis. 2022. Open-ended evolution for Minecraft building generation. *Transactions on Games* 15, 4 (2022).
[4] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. 2013. The Arcade Learning Environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research* 47 (2013).
[5] Michael Beukman, Christopher W Cleghorn, and Steven James. 2022. Procedural content generation using neuroevolution and novelty search for diverse video game levels. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM.
[6] Francesco Bodria, Fosca Giannotti, Riccardo Guidotti, Francesca Naretto, Dino Pedreschi, and Salvatore Rinzivillo. 2023. Benchmarking and survey of explanation methods for black box models. *Data Mining and Knowledge Discovery* 37 (2023).
[7] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. *arXiv preprint arXiv:1606.01540* abs/1606.01540 (2016).
[8] Jamis Buck. 2015. *Mazes for programmers: Code your own twisty little passages*. The Pragmatic Bookshelf.
[9] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. 2002. Deep blue. *Artificial intelligence* 134 (2002).
[10] Megan Charity, Ahmed Khalifa, and Julian Togelius. 2020. Baba is y'all: Collaborative mixed-initiative level design. In *Proceedings of the Conference on Games*. IEEE.
[11] Kate Compton and Michael Mateas. 2015. Casual Creators. In *Proceedings of the International Conference on Computational Creativity*. ICCC.
[12] Michael Cook. 2019. Tutorial: Generative & Possibility Space. https://www.possibilityspace.org/tutorial-generative-possibility-space/index.html. Accessed On: 18-10-2024.

[5] Available at https://kenney.nl/assets/1-bit-pack

[13] Steve Dahlskog, Julian Togelius, and Mark J Nelson. 2014. Linear levels through n-grams. In *Proceedings of the International Academic MindTrek Conference*. ACM.

[14] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. 2021. First return, then explore. *Nature* 590 (2021).

[15] Roberto Gallotta, Kai Arulkumaran, and L. B. Soros. 2023. Preference-Learning Emitters for Mixed-Initiative Quality-Diversity Algorithms. *Transactions on Games* 16 (2023).

[16] Roberto Gallotta, Graham Todd, Marvin Zammit, Sam Earle, Antonios Liapis, Julian Togelius, and Georgios N. Yannakakis. 2024. Large Language Models and Games: A Survey and Roadmap. *IEEE Transactions on Games* (2024), 1–18.

[17] Daniele Gravina, Ahmed Khalifa, Antonios Liapis, Julian Togelius, and Georgios N Yannakakis. 2019. Procedural content generation through quality diversity. In *Proceedings of the Conference on Games*. IEEE.

[18] Djordje Grbic, Rasmus Berg Palm, Elias Najarro, Claire Glanois, and Sebastian Risi. 2021. EvoCraft: A new challenge for open-endedness. In *Proceedings of the European Conference on Applications of Evolutionary Computation*. Springer.

[19] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[20] Matthew Guzdial, Nicholas Liao, Vishwa Shah, and Mark O Riedl. 2018. Creative Invention Benchmark. In *Proceedings of the International Conference on Computational Creativity*. ICCC.

[21] Matthew Guzdial and Mark Riedl. 2016. Toward game level generation from gameplay videos. In *Proceedings of the Foundations of Digital Games Conference*. ACM.

[22] Erin Jonathan Hastings, Ratan K Guha, and Kenneth O Stanley. 2009. Automatic content generation in the Galactic Arms Race video game. *Transactions on Computational Intelligence and AI in Games* 1, 4 (2009).

[23] Britton Horn, Steve Dahlskog, Noor Shaker, Gillian Smith, and Julian Togelius. 2014. A comparative evaluation of procedural level generators in the mario ai framework. In *Proceedings of the Foundations of Digital Games Conference*. ACM.

[24] Zehua Jiang, Sam Earle, Michael Green, and Julian Togelius. 2022. Learning controllable 3D level generators. In *Proceedings of the Foundations of Digital Games Conference*. ACM.

[25] Sergey Karakovskiy and Julian Togelius. 2012. The Mario AI benchmark and competitions. *Transactions on Computational Intelligence and AI in Games* 4, 1 (2012).

[26] Isaac Karth and Adam M Smith. 2017. WaveFunctionCollapse is constraint solving in the wild. In *Proceedings of the Foundations of Digital Games Conference*. ACM.

[27] Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. 2020. Pcgrl: Procedural content generation via reinforcement learning. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference*. AAAI.

[28] Ahmed Khalifa, Dan Gopstein, and Julian Togelius. 2019. ELIMINATION from Design to Analysis. In *Proceedings of the Conference on Games*. IEEE.

[29] Ahmed Khalifa, Scott Lee, Andy Nealen, and Julian Togelius. 2018. Talakat: Bullet hell generation through constrained MAP-Elites. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM.

[30] Ahmed Khalifa, Scott Lee, Andy Nealen, and Julian Togelius. 2018. Talakat: Bullet hell generation through constrained MAP-Elites. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM.

[31] Ahmed Khalifa, Diego Perez-Liebana, Simon M Lucas, and Julian Togelius. 2016. General video game level generation. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM.

[32] Hwanhee Kim, Seongtaek Lee, Hyundong Lee, Teasung Hahn, and Shinjin Kang. 2019. Automatic generation of game content using a graph-based wave function collapse algorithm. In *Proceedings of the Conference on Games*. IEEE.

[33] Raph Koster. 2013. *Theory of fun for game design*. O'Reilly Media, Inc.

[34] Antonios Liapis. 2016. Exploring the Visual Styles of Arcade Game Assets. In *Proceedings of Evolutionary and Biologically Inspired Music, Sound, Art and Design (EvoMusArt)*. Springer.

[35] Antonios Liapis. 2020. 10 Years of the PCG workshop: Past and Future Trends. In *Proceedings of the FDG Workshop on Procedural Content Generation*. ACM.

[36] Antonios Liapis, Christoffer Holmgård, Georgios N Yannakakis, and Julian Togelius. 2015. Procedural personas as critics for dungeon generation. In *Proceedings of the European Conference on Applications of Evolutionary Computation*. Springer.

[37] Antonios Liapis, Héctor P Martínez, Julian Togelius, and Georgios N Yannakakis. 2013. Transforming exploratory creativity with DeLeNoX. In *International Conference on Computational Creativity*. ICCC.

[38] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. 2013. Sentient sketchbook: computer-assisted game level authoring. In *Proceedings of the Foundations of Digital Games Conference*. ACM.

[39] Phil Lopes, Antonios Liapis, and Georgios Yannakakis. 2015. Targeting horror via level and soundscape generation. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference*. AAAI.

[40] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *arXiv preprint arXiv:1312.5602* abs/1312.5602 (2013).

[41] Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909* abs/1504.04909 (2015).

[42] OpenAI. 2024. Learning to reason with LLMs. https://openai.com/index/learning-to-reason-with-llms/. Accessed On: 24-03-2025.

[43] Diego Perez-Liebana, Jialin Liu, Ahmed Khalifa, Raluca D Gaina, Julian Togelius, and Simon M Lucas. 2019. General video game ai: A multitrack framework for evaluating agents, games, and content generation algorithms. *Transactions on Games* 11, 3 (2019).

[44] Justin K. Pugh, Lisa B. Soros, and Kenneth O. Stanley. 2016. Quality Diversity: A New Frontier for Evolutionary Computation. *Frontiers in Robotics and AI* 3 (2016).

[45] Graeme Ritchie. 2007. Some Empirical Criteria for Attributing Creativity to a Computer Program. *Minds and Machines* 17, 1 (2007).

[46] Christoph Salge, Michael Cerny Green, Rodgrigo Canaan, and Julian Togelius. 2018. Generative design in minecraft (gdmc) settlement generation competition. In *Proceedings of the Foundations of Digital Games Conference*. ACM.

[47] Noor Shaker, Antonios Liapis, Julian Togelius, Ricardo Lopes, and Rafael Bidarra. 2016. Constructive Generation Methods for Dungeons and Levels. In *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, Noor Shaker, Julian Togelius, and Mark J. Nelson (Eds.). Springer, 31–55.

[48] Noor Shaker, Miguel Nicolau, Georgios N Yannakakis, Julian Togelius, and Michael O'neill. 2012. Evolving levels for super mario bros using grammatical evolution. In *Proceedings of the Conference on Computational Intelligence and Games*. IEEE, 304–311.

[49] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529 (2016).

[50] Matthew Siper, Ahmed Khalifa, and Julian Togelius. 2022. Path of destruction: Learning an iterative level generator using a small dataset. In *Proceedings of the Symposium Series on Computational Intelligence*. IEEE.

[51] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. 2018. Procedural content generation via machine learning (PCGML). *Transactions on Games* 10, 3 (2018).

[52] Julian Togelius and Jurgen Schmidhuber. 2008. An experiment in automatic game design. In *Proceedings of the Symposium On Computational Intelligence and Games*. IEEE.

[53] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron Browne. 2011. Search-based procedural content generation: A taxonomy and survey. *Transactions on Computational Intelligence and AI in Games* 3, 3 (2011).

[54] Ruben Rodriguez Torrado, Ahmed Khalifa, Michael Cerny Green, Niels Justesen, Sebastian Risi, and Julian Togelius. 2020. Bootstrapping conditional GANs for video game level generation. In *Proceedings of the Conference on Games*. IEEE.

[55] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* 575 (2019).

[56] Vanessa Volz, Jacob Schrum, Jialin Liu, Simon M Lucas, Adam Smith, and Sebastian Risi. 2018. Evolving Mario levels in the latent space of a deep convolutional generative adversarial network. In *Proceedings of the Genetic and evolutionary computation conference*. ACM.

[57] Ziqi Wang, Jialin Liu, and Georgios N Yannakakis. 2022. The fun facets of Mario: Multifaceted experience-driven PCG via reinforcement learning. In *Proceedings of the Foundations of Digital Games Conference*. ACM.

[58] Georgios N Yannakakis and Julian Togelius. 2011. Experience-driven procedural content generation. *Transactions on Affective Computing* 2, 3 (2011).

[59] Georgios N Yannakakis and Julian Togelius. 2014. A panorama of artificial and computational intelligence in games. *Transactions on Computational Intelligence and AI in Games* 7, 4 (2014).

[60] Yifan Yao, Jinhao Duan, Kaidi Xu, Yuanfang Cai, Zhibo Sun, and Yue Zhang. 2024. A survey on large language model (LLM) security and privacy: The good, the bad, and the ugly. *High-Confidence Computing* (2024), 100211.


## Appendix A Experiments with LLM Generators

With the surge of LLMs [60], we wanted to showcase that the PCG Benchmark can be used to compare LLM generators [16] to more classic approaches. We compared 100 separate runs of the provided algorithms (Random, ES, and GA) with a constructive generator [47] and two few-shot LLM generators based on *Llama 3.2* and *DeepSeek-r1*. We tested them on only 3 simple problems of the PCG Benchmark: Binary, Sokoban, and Zelda. The constructive

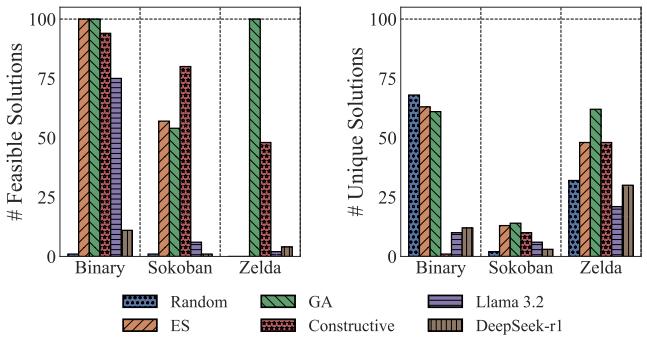


**Figure 4: The number of feasible and unique solutions over 100 separate runs on Binary, Sokoban, and Zelda using six different methods (three search-based generators, one constructive generator, and two few-shot LLM generators).**

generator starts by building a 2D maze using Prim's algorithm [8]. This generated maze is used as-is for Binary. For Sokoban and Zelda, the script erases more than 50% of the walls to allow for open areas, then adds the missing objects in the level at random locations (for Sokoban, these locations are restricted such that crates have no more than one side blocked). LLM generators use a simple prompt that explains the goal of the game and how to play it, followed by the goal of the generator and five example levels.

Figure 4 shows the comparison between these algorithms from the perspective of quality (number of feasible solutions) and diversity (number of unique solutions) over 100 runs, extending the findings from Fig. 3. GA has more feasible solutions overall, although the constructive algorithm surpasses it in Sokoban. In Sokoban, the script is fairly thorough (e.g. constraining where crates can be placed) and thus it is not surprising that it can generate many feasible solutions. It is worth noting, however, that most of these solutions are not unique: GA and ES find more unique feasible results even if the number of feasible results is fewer than for the constructive method. From LLM methods, *Llama 3.2* outperforms *DeepSeek-r1* in terms of quality for Binary and Sokoban, but both LLMs find very few feasible solutions in Zelda. This disparity was surprising because reasoning models such as *DeepSeek-r1* usually perform better on language tasks than traditional models such as *Llama 3.2* [42]. Looking at the generated examples, we noticed that *Llama 3.2* has a higher chance of copying some of the examples used in the prompt. It seems that the added reasoning tokens [19] in *DeepSeek-r1* might have pushed it towards understanding the examples in the prompt and trying to create new ones; however, its results were often infeasible. This first, preliminary experiment showcases how the PCG Benchmark can be used for current and emerging technologies such as LLM-based generation [16].