Agentic PCG: Procedural Content Generation via Tool-using LLMs

Zehua Jiang
Game Innovation Lab
New York University
Brooklyn, New York
zehua.jiang@nyu.edu

Sam Earle
Game Innovation Lab
New York University
Brooklyn, New York
sam.earle@nyu.edu

Ahmed Khalifa
Institute of Digital Games
University of Malta
Msida, Malta
ahmed@khalifa.com

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

***Abstract*—We propose *Agentic Procedural Content Generation*, a paradigm in which tool-calling large language models generate video game levels that are both functional and steerable via natural language. Equipped with local brushes, generative algorithms, and evaluation functions, agents iteratively refine content toward specified design objectives. We show that the framework applies to both static level design tasks and environments with dynamic gameplay mechanics, while reliably satisfying diverse functional constraints. Beyond satisfying such constraints, the same framework can also shape more open-ended aspects of gameplay, enabling agents to optimize not only for functionality but also for richer player experiences.** [1]

***Index Terms*—Large Language Models, Procedural Content Generation**

I. Introduction

The advent of agentic AI implemented as harnesses on large language models has prompted many to rethink their processes. Workflows, habits, and whole applications are reimagined in the form of reasoning LLMs that call tools to probe and poke at the world. How about content production pipelines?

A large variety of games, including but certainly not limited to the multitude of roguelikes and roguelites, feature procedural generation of some aspect of game content [1]. Generally speaking, the PCG solution is handcrafted for each game, although general algorithmic strategies are reused. There is also a blossoming field of PCG research in academia, where various such algorithmic strategies are developed. These strategies vary widely, but many fall into the wider generate-and-test paradigm, where editing actions are interleaved with automated evaluation of the generated content.

What we propose here is to use the wide variety of methods developed within PCG as building blocks for LLM-driven agentic workflows. This amounts to deconstructing existing PCG methods and reusing both their editing and evaluation parts as tools that can be called by a properly prompted LLM. The promise of this approach is to make it much easier to create content generators for specific games while reusing existing code. We call this approach Agentic Procedural Content Generation.

[1]The project webpage is available at https://jiangzehua.github.io/AgenticPCG/.

The bulk of this paper describes a system in which a powerful LLM is given access to a number of tools for editing and evaluating levels, and given prompts to generate them. To show the versatility of our approach, we generate levels for Binary Maze, Super Mario Bros, Zelda, Lode Runner, and Sokoban. The environments are mainly taken from the PCG benchmark [2], which provides a standard API for evaluation functions.

A key concern for most game developers is that the method runs fast and on device, as the economies of most games do not permit querying frontier models at runtime. To this end, we investigate the capability degradation of Agentic PCG when using local LLMs in the orchestrating role. By grounding agents with domain-specific tools and evaluation metrics, we enable models with varying capacity to iterate productively.

Our key contribution is a unified tool-using LLM framework for procedural content generation. First, we show that the framework is effective in both static design settings and environments with dynamic gameplay mechanics. Second, the framework is not restricted to primitive edit actions such as tile placement: it can also integrate traditional PCG methods as callable tools for level editing. Third, it supports open-ended natural language requirements in addition to explicit functional constraints, enabling joint control of both measurable design targets and broader gameplay qualities.

II. Related Work

While many content generators deployed in games use ad hoc constructive approaches, the research field around PCG has mostly focused on approaches based on optimization and/or machine learning.

Search-based procedural content generation (SBPCG) [3] uses a search algorithm to search the space of possible content with the help of an evaluation function. The evaluation function can be as simple as evaluating statistics about the current content (direct evaluation) or a simulation-based evaluation in which an AI agent uses the content during play (e.g. playing through a level) and evaluation is applied to the resulting playtrace. SBPCG has been used to create various types of content, such as 3d models [4], tutorials [5], levels [6], rules [7], and full games [8]. Procedural content generation via machine learning (PCGML) [9] uses input examples instead of

an evaluation function to learn how to generate content. Instead of generating the content directly, machine learning is used to train a generator that can produce content that is similar to, yet distinct from, the input examples. Various generative models have been used, including N-Grams [10], Markov Chains [11], LSTMs [12], AutoEncoders [13], and GANs [14].

With the rise of Large Language Models (LLMs), researchers have explored the training and usage of LLMs to generate different types of content, such as game levels [15]–[17], game maps [18], game mechanics [19], and puzzle games [20]. One of the biggest advantages of using LLMs is the capacity for designers to express specifications for the generated content using natural language, enabling easier and more flexible interaction with the system.

Gallotta et al. [21], [22] investigate the use of natural language within a mixed initiative system [23], probing its ability to act as a conduit for capturing designer intent. Instead of modifying actions by hand, the designer can directly express a high-level plan or design goal in natural language, which an LLM then converts into multiple tool calls (e.g. creating a room, adding enemies). The results show that decomposing large tasks into smaller ones via multiple LLM calls is more effective than asking the system to make large changes in a single pass. Our work expands on Gallotta et al.'s framework by allowing the LLM to be an agentic system in which the LLM can not only call tools to directly edit the content, but also nested PCG functions. We also explore the LLM as a fully autonomous system as opposed to a mixed initiative system.

Even with all their recent advancements, LLMs are still prone to hallucinations and mistakes. Generating functional content such as game levels (that not only need to look good but also be playable and solvable) is a very hard task for LLMs as they cannot easily capture the spatial connections between different game tiles in the level. LLMs also often have problems correctly counting the number of objects in the level, and they cannot simulate the game to make sure that it is playable. These problems call for using external tools, both to inform the LLM and to allow it to affect the right kind of changes to the level.

A major inspiration for this work is the PCGRL line of research [24]–[28], which formulates level editing as a reinforcement learning problem: an agent traverses the level, selects local edit actions at each position, and learns a controllable and robust policy from rewards defined by level evaluation metrics. Similarly, we cast level design as an iterative, RL-like task, with a reward function defined by distance from a set of target values over a number of evaluation metrics. The agent's "action space", however, is much vaster in our work, in which LLMs effectively write code to call arbitrary level-editing tools.

III. Methods

We propose an LLM-based agent framework for procedural content generation (PCG) that formulates level design as iterative search-based optimization over a discrete tile grid. Rather than having the LLM directly generate the level, we host the game environment as a reinforcement learning environment in which all game dynamics are encapsulated and managed. At each step, the LLM, acting as an agent, operates in a closed loop: it observes the current level state and evaluation metrics, reasons about what to improve, invokes structured tool calls to edit the level, and then the system decides whether to accept or reject the proposed changes based on a loss function. This cycle repeats until a change budget is exhausted or the agent elects to stop.

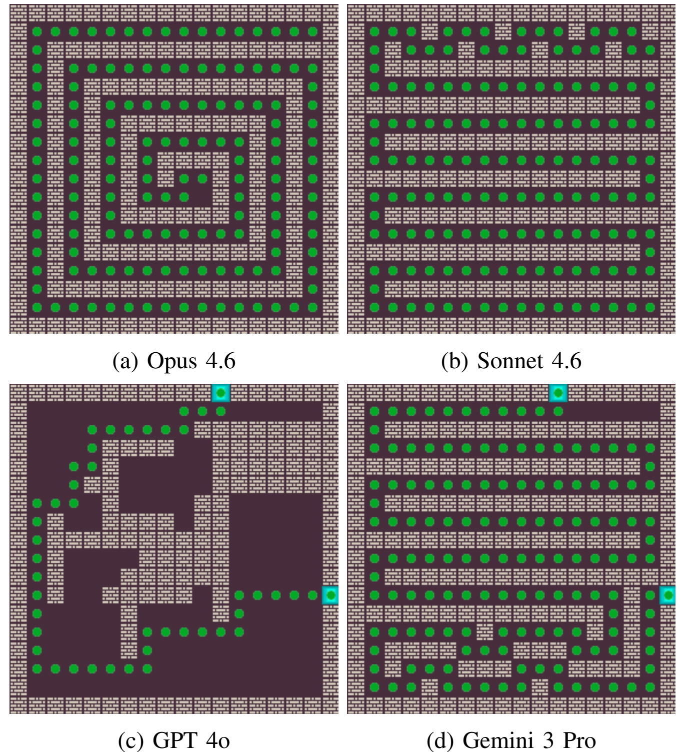


Fig. 1: **Binary** and **Binary Door**. Output of LLMs instructed to optimize quality given default target metrics. The tooth/merlon-like pattern adorning the zig-zags (right) commonly recurs as a means of squeezing out more path length. The optimal strategy in the Binary Door pattern (bottom) is to adapt zig-zags to connect with doors, but 4o fails in this trial.

A. Environment

We adopt the PCG Benchmark [2] as the evaluation and problem definition backend for our experiments. Our proposed framework is tested in six different problems:

- **Binary** is a 2D maze of solid and empty tiles which must fully connected, and in which the longest shortest path between any two empty tiles must be more than 20. (A)
- **Binary Door** is similar to the binary problem, but the distance is measured between start and end "door" tiles along the edge of the level. (B)
- **Zelda**, based on [29], is a simple top-down dungeon crawler in which the player needs to be able to grab a key and reach the exit while killing enemies. Levels

should have one player, one key, one door, a minimum of 3 enemies, and a solution length of 18. (C)

- **Lode Runner**, based on [30], is a platformer puzzle game. The player controls a character that can move left and right and climb ladders but cannot jump. The goal is to collect all the gold without being arrested by the enemies. Levels for this game should have one player, at least 3 enemies, at least 6 gold, and there should be a path from the player tile to all the gold (as measured by a simplified pathfinding agent assuming static enemies). Finally, the gold should be distributed such that the player needs to explore some percentage of the level in order to reach it. (D)
- **Sokoban**, based on [31], is a top-down grid-based puzzle game. The player needs to push all the crates in the level to cover a specific target. Levels should have one player, at least one crate, and be solvable using an A* agent in more than 10 steps. (E)
- **Super Mario Bros.**, based on [32], is a platfomer game. The player needs to traverse the level from left to right while avoiding falling into pits and avoiding enemies or killing them by jumping on top of them. Levels should have no floating enemies, and be solvable by an A* agent or a greedy agent that is always trying to jump and move right. (F)

These problems were selected as they span a range of complexity, from simple mazes to full platformer physics simulations. The level construction requirements are also increasing in difficulty, from simple tile counts and static pathfinding, to puzzle games with mutable states (requiring tree search), and dynamic environments with moving enemies. See Section A for full details about the setup of each problem, representation, and target metrics.

B. Available Tools

Our LLM agent edits levels via a structured set of tools rather than directly generating the entire map as raw tokens. These tools define a constrained and interpretable action space that includes both low level edit operations and higher level procedural generators. This design improves controllability and makes the generation process more transparent: agents make plans and provides explicit rationale before taking action, and each action corresponds to a human readable operation such as placing a tile, drawing a wall segment, filling a region, or applying a procedural generator.

1) Basic Edit Tools: The basic tile edit tool supports direct modifications to the current level. It can place a single tile, draw a line, or fill or outline a rectangular region, and is available by default in all domains.

2) Classical PCG Tools: We also incorporate classical procedural content generation operators as callable tools with detailed descriptions. These include:

Generation Tools:

- `random`: Generates a completely new random binary layout, ignoring the current level. Each tile is independently set to wall with probability 0.5.

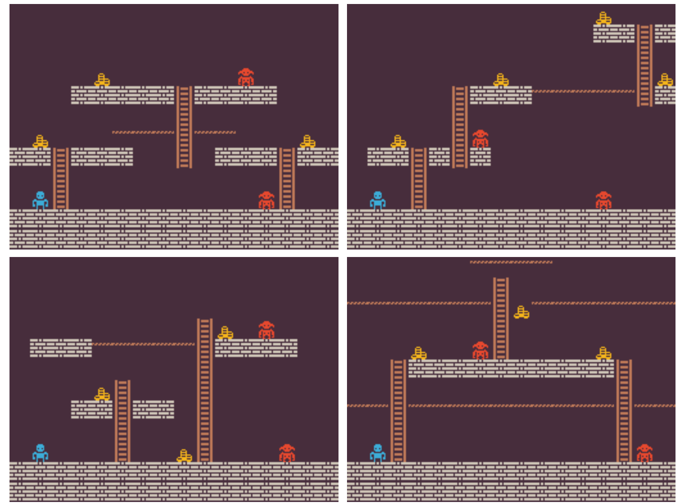


Fig. 2: **Lode Runner**. Gemini 3 Pro produces small but serviceable final Lode Runner levels. The central ladder in the upper left originally reached to the ground, but lower rungs were removed to satisfy the *n. ladders* constraint.

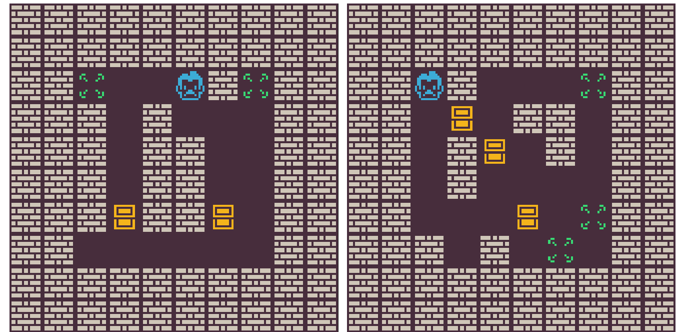
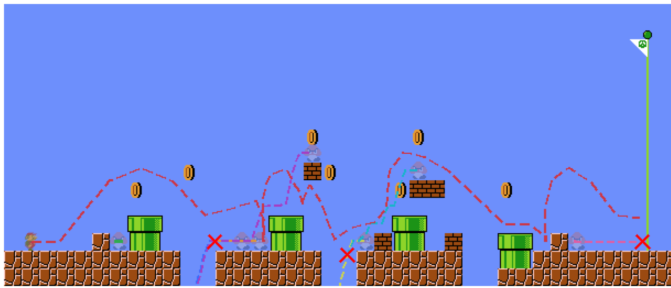
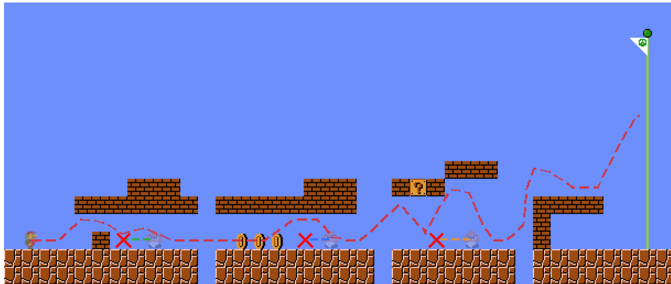


Fig. 3: **Sokoban**. Gemini 3 Flash produces compact final Sokoban layouts that satisfy the crate and player count constraints. The resulting levels are cramped, demanding careful manoeuvering from the player, with crates and targets arranged in simple enclosed rooms separated by walls.

- `maze`: Carves a perfect maze into the existing all-wall tiles using depth first search. Only converts walls to empty but never adds walls. Produces long, winding single-tile-wide corridors with no loops in which every pair of empty tiles is connected by exactly one path.
- `bsp`: Generates a room-and-corridor layout using Binary Space Partitioning. Recursively splits space into rectangles, carves a room in each partition, connects adjacent rooms with corridors. More splits means more smaller rooms with longer paths. Defaults to 3 recursive splits with minimum partition height and width of 5.
- `digger`: Generates a cave-like layout using a random-walk digger. Starts from an all-wall grid, walks randomly carving empty tiles, occasionally carving rooms. Stops when empty fraction reaches the pre-set stop size, which defaults to 0.3.
- `wfc`: Generates a maze using Wave Function Collapse



(a) o3 mini.



(b) Gemini 2.5 Pro

Fig. 4: **Super Mario Bros**. The red dashed curve shows the simulated agent path; each colored trajectory shows a different Goomba's motion, with transparent sprites indicating their initial positions. Red crosses mark the locations where Mario stomps and kills a Goomba. Generators tend to prefer fairly linear levels (ensuring solvability by a resource-constrained A* player agent), with some verticality and/or pits to force *n. jumps* from the player. Goombas are placed strategically—on unavoidable precipices or in cramped tunnels—to ensure satisfaction of *n. kills*.

from a reference image. Infers local constraints from the reference and produces a consistent output. Can fail if constraints become unsatisfiable. The level will be left unchanged on failure.

Refinement Tools:

- `ca`: "Smooths" the current level in place using a cellular automaton. Removes isolated wall/empty tiles. Turns noisy layouts into organic cave-like shapes. No effect on uniform levels (all-empty or all-wall).
- `connect`: Post-processes the current level to ensure connectedness. Finds all connected empty regions, fills regions of less than 5 tiles with wall, then connects remaining regions with straight corridors.

The generation tools generate or override a new level from scratch, while refinement tools modify the existing level in place. Consequently, effective level optimization often requires composing multiple tools rather than applying a single operation in isolation. For instance, if the level is initialized as all-empty, applying `ca` directly is typically ineffective, since cellular automata rely on existing local variation to produce meaningful structure. A more effective strategy is to first apply `random` to introduce heterogeneous patterns, and then use `ca` to smooth the layout into a more coherent form.

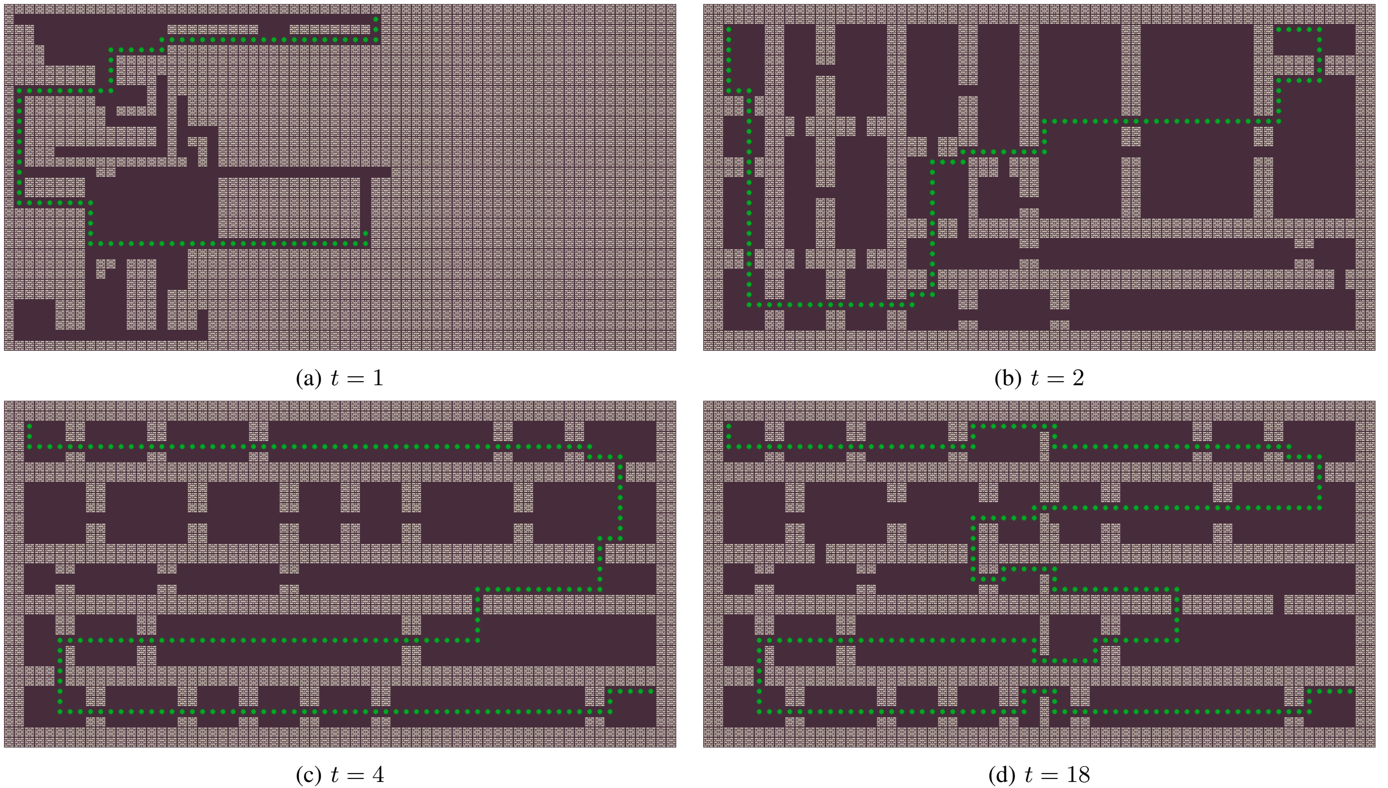
TABLE I: Cross-model comparison of quality, diversity, and controllability scores across five game domains. Best scores per game are in **bold**.

Game	Model	Qual ↑	Divers ↑	Control ↑
Binary	Claude Sonnet 4.6	0.9970	0.1500	0.9459
	Claude Opus 4.6	**1.0000**	0.1000	0.8319
	Gemini 2.5 Pro	**1.0000**	0.3684	0.9939
	Gemini 3 Flash	**1.0000**	0.1500	**1.0000**
	Gemini 3 Pro	**1.0000**	0.1176	**1.0000**
	GPT-4o Mini	0.8329	**0.4500**	0.6566
	o3-mini	0.9939	0.3000	0.9008
	Qwen 3.5 35B	0.9903	0.3500	0.7718
Binary Door	Claude Sonnet 4.6	0.9805	0.5000	0.6943
	Claude Opus 4.6	0.9687	0.4000	0.7189
	Gemini 2.5 Pro	**1.0000**	0.3333	0.7372
	Gemini 3 Flash	**1.0000**	0.2500	0.8079
	Gemini 3 Pro	**1.0000**	0.1538	**0.8186**
	GPT-4o Mini	0.8867	0.2500	0.4931
	o3-mini	0.9330	**0.5714**	0.5825
	Qwen 3.5 35B	0.8516	0.2778	0.4251
Lode Runner	Claude Sonnet 4.6	0.2344	0.0588	0.9898
	Claude Opus 4.6	0.2292	0.0500	0.9975
	Gemini 2.5 Pro	0.2292	0.0500	0.9979
	Gemini 3 Flash	**0.2615**	0.0500	**1.0000**
	Gemini 3 Pro	**0.2615**	0.0500	**1.0000**
	GPT-4o Mini	0.2292	**0.1250**	0.9992
	o3-mini	0.2292	0.0588	0.9982
	Qwen 3.5 35B	0.2284	0.0345	0.9987
SMB (A*)	Claude Sonnet 4.6	0.9975	0.8500	0.8061
	Claude Opus 4.6	0.9968	0.8500	0.7805
	Gemini 2.5 Pro	0.9978	0.8500	0.8115
	Gemini 3 Flash	0.9973	0.9500	0.7568
	Gemini 3 Pro	0.9977	0.9500	**0.8262**
	GPT-4o Mini	**0.9988**	**1.0000**	0.7665
	o3-mini	0.9955	0.9286	0.8148
	Qwen 3.5 35B	0.9948	0.9375	0.7933
SMB (Greedy)	Claude Sonnet 4.6	0.9978	0.9000	0.9713
	Claude Opus 4.6	0.9975	0.8500	0.9464
	Gemini 2.5 Pro	0.9979	0.8500	0.9701
	Gemini 3 Flash	0.9971	0.9000	**0.9878**
	Gemini 3 Pro	0.9976	0.9500	0.9760
	GPT-4o Mini	**0.9990**	**1.0000**	0.9511
	o3-mini	0.9951	0.9375	0.9437
	Qwen 3.5 35B	0.9967	0.8182	0.9176

C. Optimization Loss

The optimization loop follows a hill-climbing strategy in which the LLM agent iteratively proposes edits to a game level, and each candidate is evaluated by the PCG benchmark to produce a scalar score. The score is a weighted sum of 4 problem-specific components: controllable metric alignment, solvability, structural validity, change penalty. Table II summarizes the scoring components for each problem type.

For each controllable metric, the designer can specify whether the generated level should match some target value, or maximize the value. The target matching term penalizes deviation from the desired value as $-w \cdot |x - x^*|$, while a



(a) $t = 1$

(b) $t = 2$

(c) $t = 4$

(d) $t = 18$

Fig. 5: Iterative level editing with PCG tools. The agent calls a Binary Space Partitioning algorithm on steps 2 and 4, then tweaks the layout with small alternate walls and corridors to optimize path length. The call to BSP on step 4, which radically overwrites the existing level, is accepted because it greatly increases the level's functional score.

maximization term rewards higher values as $+w \cdot x$, where w is the weight assigned to the metric, x is its current value, and x^* is its target value. A solvability term applies a large bonus or penalty depending on whether the level is playable. And a change penalty discourages excessive modification by penalizing candidates that differ from the last accepted level by more than 60% of tiles. Structural validity terms enforce hard domain-specific constraints, such as special entity counts that are prerequisites for a playable level but are not designer-controllable parameters. These terms apply a fixed bonus when the constraint is satisfied and a penalty proportional to the deviation otherwise. Not all problems have structural validity terms.

At each step, the candidate is accepted if its score exceeds that of the current level. When the score is worse, the system supports two exploration strategies: simulated annealing, which accepts inferior candidates with probability $p = \exp(\Delta s/T)$ where Δs is the percent of tiles changed relative to the previous level, and the temperature $T = T_0 \cdot \alpha^t$ decays over steps; and epsilon-greedy, which accepts worse moves with a fixed probability ϵ. If no tiles were changed by the proposed edits, the candidate is unconditionally rejected.

D. LLM Response

In each optimization iteration, the LLM agent communicates through a structured JSON protocol. The agent receives the current level state, evaluation metrics, and feedback from prior steps, then responds with a step message containing rationale, a plan, and a list of tool calls to edit the level, or with a stop message to terminate early. The optimizer executes the tool calls, evaluates the resulting level, and accepts or rejects the candidate based on the scoring function.

When configured with a conversation window of length $k > 1$, the agent receives the previous $k-1$ prompt–response pairs augmented with additional feedback, namely the accept/reject outcome and reason, per-metric deltas between consecutive steps, and a summary of tool call success rates and tiles changed. This enables the agent to learn from its own recent trajectory without requiring explicit fine-tuning.

The optimization terminates when the cumulative number of accepted tile changes reaches a budget of $m \cdot H \cdot W$ (where m is a configurable multiplier, and H and W are grid dimensions), when the agent outputs a STOP message, when a maximum step count is reached, or when a circuit breaker triggers after consecutive LLM errors.

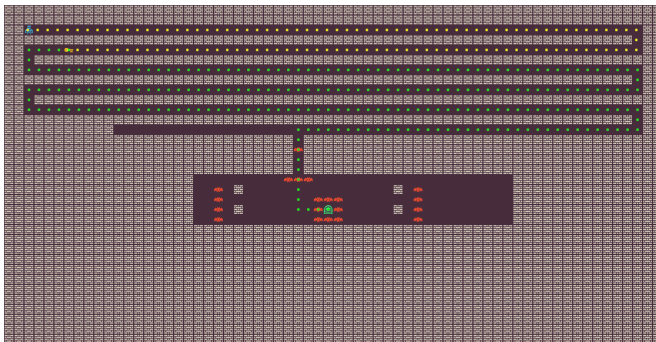
IV. Experiments

We conduct our experiments using eight LLM backbones from four model families, varying in size and release date: Claude Sonnet 4.6 [33] and Claude Opus 4.6 [34] (Anthropic); Gemini 2.5 Pro [35], Gemini 3 Flash preview [36], and Gemini 3 Pro preview [37] (Google); GPT-4o-mini [38] and

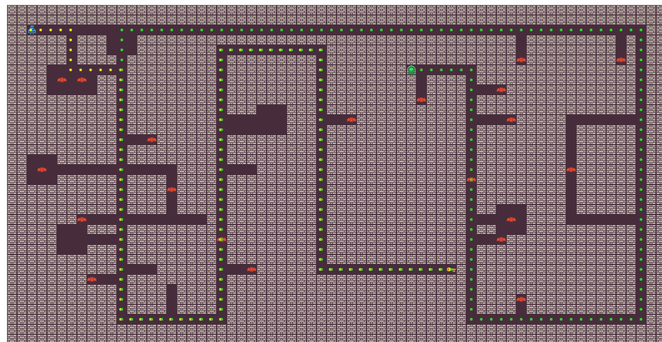
(a) ***Goomba Cliff Dive***. *"Make a level in which Mario jumps off a cliff and bounces off a series of falling Goombas."* Though the agent follows the prompt by making Mario bounce off a series of falling Goombas, the setup only partially supports the intended bounce chain.

(b) ***Goomba Household*** *"Make a level that includes a cross-section of a two-storey Goomba home. Have two Goombas around a table on the main floor."* Indeed, two Goombas appear seated at either side of a wall block on the main floor.

(c) ***Goomba Cave***. *"Underground cave filled with Goombas."* The agent takes a resourceful shortcut, placing a roof in the sky which does not affect gameplay in the otherwise linear and generic level.



(d) ***Challenge Room*** *"Generate a level with a challenge room."* The agent creates a winding corridor (to match the path length target constraints), leading to a distinct enclosed room densely populated with enemies (from free-form language instruction), functioning as a dedicated challenge area before the door.



(e) ***Branching Path*** *"Generate a level with a branching path."* The agent produces a maze-like dungeon with multiple interconnected corridors and decision points, creating genuine branching paths between the player, key, and door.

Fig. 6: Agent LLM level generators attempt to follow freeform instructions while satisfying functional constraints. Prompts have non-negligible, interpretable effects on final levels, but agents prioritize their functional targets, which can lead to simplified

o3-mini [39] (OpenAI); and Qwen 3.5-35B-A3B-FP8 [40] (Alibaba), served locally via vLLM on two NVIDIA 4090 GPUs, each with 24GB VRAM.

We evaluate the levels from three different perspectives: *quality*, *controllability* and *diversity*, following the latest version of the PCG Benchmark [2]. Quality score is the average score of each metric given a predefined fixed target, measuring the ability of the agent in creating levels that satisfy hard-coded functional constraints, providing a consistent challenge. For each of the metrics in a certain game, we calculate a trapezoidal function returning 0 at the edges, linearly ramping up/down, until reaching the fix target, and 1.0 on the plateau. By doing these, we normalize the metric score with different ranges. And the final quality score takes the average of all the metrics of interest in a given game. Controllability score follows a similar calculation pattern, with different targets sampled from the target metric space and add/subtract with an error tolerance term to form the plateau of trapezoid function. Diversity scores derive from the comparison of pairs of levels using measures based on the mechanics of each game. Some use structural difference like hamming distance, others use behavioral difference such as solution paths, exploration maps, or visited tiles. Note that the evaluation score is separate from the optimization score used to admit agent actions during the generation process, and is computed only on the final optimized level. Although our framework supports all six environments, we conduct the experimental sweep on four representative domains: Binary, Binary Door, Lode Runner, and Super Mario Bros. with both A* and greedy agents. In quality experiments, we conduct 20 trials with controlled random seeds and fixed challenging target metrics. Quality scores are averaged across trials, and diversity is computed pairwise across all solvable final levels. In controllability experiments, we conduct 10 trials with randomly sampled control targets per trial,[2] and average scores over solvable final levels. Trials that fail due to consecutive server-side API errors or timeouts are discarded.

V. Results

All models achieve reasonably strong performance under the PCG Benchmark evaluation metrics, indicating an ability to perceive map structure, reason about optimization opportunities, and generate effective, feasible editing plans (Table I). This is notable given that the models receive no multimodal input and instead operate solely on flattened one-dimensional textual representations of the level. We further observe that

[2]All randomness is seeded, so the sampled control targets are the same across models.

Problem	Controllable Metrics	Solvability	Structural / Additional Terms
Binary	Longest shortest path length; number of connected regions	Connected empty tiles exist (path length > 0)	—
BinaryDoor	Path length between doors; number of connected regions	Doors are connected via doors	—
Zelda	Number of enemies; player-to-key path length; key-to-door path length; total solution length	Player can reach the key and the key can reach the door	Exactly one player, one key, and one door; exactly one connected region
Sokoban	Number of crates; number of target tiles; solution length	Automated solver finds a valid solution	Exactly one player; number of crates equals number of targets
LodeRunner	Number of gold, enemy, ladder, and rope tiles;	Exactly one player, at least one gold, and all gold reachable; penalty graduated by fraction of uncollectable gold	Exactly one player
SMB	Enemies killed; coins collected; jumps performed (all from gameplay simulation)	Simulated agent reaches the end of the level; scored continuously as completion fraction	No malformed tube formations

TABLE II: Problem-specific controllable metrics, solvability requirements, and additional structural constraints used in our *optimization loss* function (not evaluation metrics like quality, diversity and controllability scores). Additionally all problem types share a graduated change penalty that activates when more than 60% of tiles differ from the last accepted level, scaling linearly with the excess change ratio.

smaller local models can still function effectively within our framework on benchmark-based objectives: for example, Qwen3.5-35B-A3B is able to follow the loss-based optimization harness, adhere to the required JSON output format, and optimize toward the selected target metrics. However, larger frontier models more consistently produce levels that are stronger along less explicitly specified dimensions, such as playability, interestingness, and aesthetic quality.

In the Binary domain (Fig. 1), all the large frontier models are able to learn the optimal (but degenerate) strategy—a zigzag. The also-optimal spiral pattern is less common, perhaps because it is more complicated to express as a series of walls. The Binary Door domain adds marginal difficulty to the task, though an effective strategy involves small adaptations of zigzags to accomodate variable door placement along the edges of the map.

The agents also have a relatively easy time meeting most of the functional constraints in Lode Runner, namely the reachability of the gold, though quality scores suffer due to divergence from solutions generated for a human dataset of levels, which is weighted relatively heavily.

In Super Mario Bros., models are capable but unambitious. They tend to build flat, low levels, using bottomless pits to force jumps. They can be quite precise in forcing the A* agent to kill a certain number of enemies, even working around the agent's heuristic repulsion from close quarters with them. Indeed, they often build ceilings leaving no option of comfortably clearing an enemy by jumping over it, or time an enemy's walk or even mid-air fall such that the player bounces off of it at a particular position.

VI. Discussion

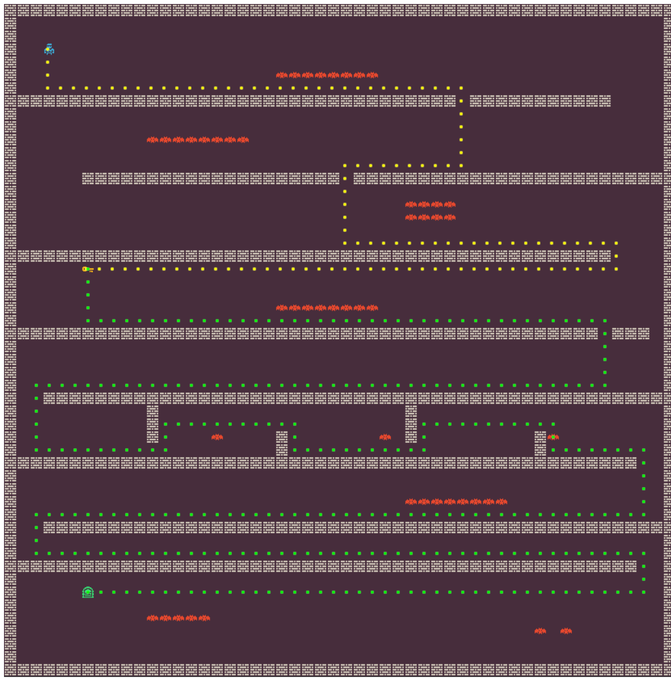
We run extensive quantitative benchmarking on PCG Benchmark, but find that most LLM generators are able to easily saturate these evaluation scores (e.g. quality score is often 1.0 on binary domains). However, the task becomes substantially more difficult in domains with richer game mechanics and stricter functional constraints. For instance, in a Mario level of width 32, generating a playable map that induces 5 jumps and 3 kills is already a nontrivial design problem. Still, Agentic PCG feels much more efficient and effective than prior means of optimizing content for functional constraints [24].

Perhaps this is because, intuitively, an optimal game designer in our toy setting probably needs to be internally modelling the game engine (e.g. enemy movements) and player to do well. Such behavior seems at least plausible to arise from LLMs when engaging with the task, in contrast to e.g. small RL models that trained on it exclusively and which may merely be memorizing mappings of level states to edit actions. Instead, LLMs engage in an explicit process of iterated trial and error with the PCG Benchmark, trying to predict its dynamics in order to achieve target metrics.

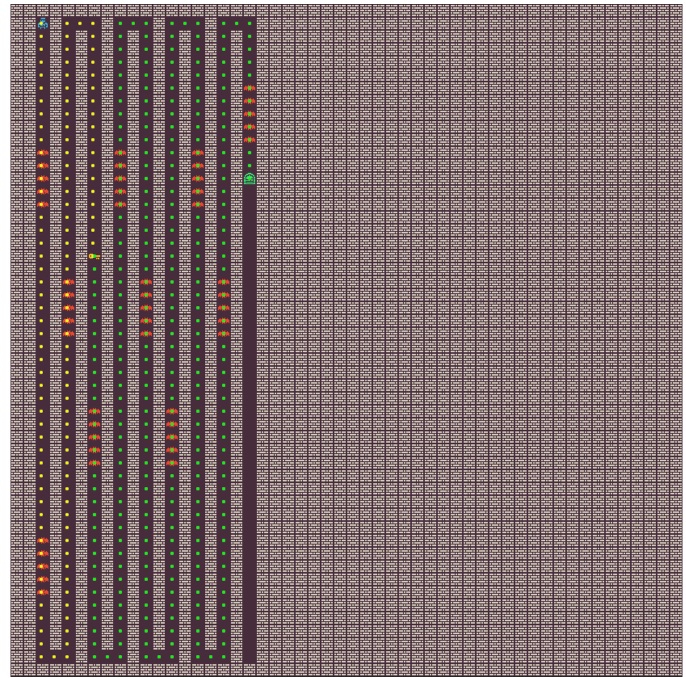
Separately, we note at least one aspect of the PCG Benchmark which is limiting or confounding in the context of LLM generators. Specifically, the Lode Runner environment penalizes levels that generate player solution paths that diverge from those found in a sparse human dataset. The motivation behind this metric is to bias environment generation toward human priors and notions of interestingness, but it would seem like a relatively weak signal toward this end as compared to the potential expertise of an LLM trained on scores of gameplay-related data (Lode Runner included), or even the specific, potentially idiosyncratic instructions from its (human) prompt-writer.

A. Different Edit Preference of Different Models

Although Table II captures the numerical outcomes of optimization, these aggregate metrics alone do not fully reflect the differences in model behavior. Game level design is not



(a) Gemini 2.5 Pro on the 50×50 Zelda task.



(b) Gemini 3 Flash on the 50×50 Zelda task.

Fig. 7: Example comparison of two LLM agents on the same 50×50 Zelda optimization problem with identical all-empty initialization and target metrics. Although both runs satisfy nearly the same functional objectives, they exhibit markedly different level editing behaviors and optimization trajectories.

merely a process of matching functional constraints, and different LLM backbones can exhibit clearly different editing patterns and preferences when performing level construction and refinement.

Figure 7 provides an illustrative example, showing that, even under the same problem setting, different LLM backbones can produce substantially different level editing behaviors that are not immediately apparent from aggregate numerical results alone.

For the 50×50 Zelda task, the two agents reach nearly the same final outcome in terms of functional target matching, but through very different editing behaviors and with totally different final level layouts. Gemini 3 Flash reaches the maximum score of 750 in only 5 steps, with 3 accepted edits (60% acceptance rate), whereas Gemini 2.5 Pro required 100 steps to reach 749, with 8 accepted edits (8% acceptance rate). The main difference lies in edit scale: Gemini 3 Flash performs a large initial rewrite, changing $2,075$ tiles (83% of the map) and improving the score from -1172 to 377 in one step, then reaches the optimum with a small 20-tile refinement. In contrast, Gemini 2.5 Pro improves the level more gradually, with its largest accepted edits changing only about 323–376 tiles. Tool usage shows a similar pattern: Gemini 2.5 Pro makes 269 total tool calls (178 line, 88 single, 2 rect.), while Gemini 3 Flash uses only 30 (21 line, 7 single, 1 rect.). Although both agents primarily rely on line-based tile placement, Gemini 2.5 Pro uses substantially more tool calls and improves the level gradually through smaller local modifications, whereas Gemini 3 Flash uses far fewer calls and benefits from a single large rectangular edit that rapidly imposes global structure.

B. Balancing Functional Constraints and Open-Ended Design Intent

An LLM is already able to act as an optimizer in our framework. However, within the hill-climbing loop, whether a proposed edit is accepted still depends solely on the optimization loss, through direct comparison of objective values. This loss is derived only from user-specified target-metric matching constraints, and therefore captures only a limited subset of the design objectives; it does not even include all computable metrics. As a result, the LLM's own design priors and intuitions may not align with a loss function defined over only a selected set of metrics. Nor are the LLM's design intuitions guaranteed to coincide with the goals of the game designer using the system. This makes it difficult to determine which optimization strategy is truly preferable. Using the LLM itself as the optimization judge may be overly loose, and may place disproportionate weight on free-form language control, which LLMs naturally handle well. They can often interpret open-ended design instructions without explicit metric computation. In contrast, to reason reliably about selected functional constraints, the model must depend on tool-based evaluation of metrics. Yet if optimization is driven too strongly by functional-constraint matching alone, the search process may become overly rigid, forcing compromises that

prevent open-ended language instructions from being fully satisfied. Balancing metric-based constraint satisfaction with free-form design intent remains an important direction for future research.

VII. Conclusion

Relative to prior work that trained or fine-tuned (L)LMs to generate grid-based levels tile-by-tile [15], [16], our agentic workflow seems to be capable of generating more complex levels that consistently meet both quantitative and qualitative metrics. We find that recent frontier reasoning models (along with their smaller open-source cousins) can adapt to various domains by leveraging existing PCG tooling [2] to understand the functional, gameplay-oriented outcome of their actions. They can use existing PCG algorithms (binary space partitioning, cellular automata, search), and lower-level controls resembling "brushes" typical of human-facing level editors (tile and wall placement, flood fill) to explore the space of levels competently.

Most importantly, LLM agents can scaffold the given evaluation metrics to follow higher-order functional constraints. They can build dungeon levels with certain patterns of rooms and corridors, and generate gameplay-relevant features like challenge rooms (Fig. 6d) and branching pathways (Fig. 6e). They can optimize not only the search-based proxy for difficulty provided to them as a numeric value, but also affect relevant but more open-ended properties of gameplay, e.g. designing levels that force players to certain regions or to double back due to structural barriers. Future work should explore formally encoding this process of scaffolding by giving agents the ability to synthesize new evaluation and design tools in response to free-form user objectives [41].

A key advantage of our design is that it allows traditional PCG methods to remain useful within an LLM-based framework. Levels that only optimize functional constraints can be correct but still appear unnatural or uninteresting. By exposing classical generators as tools, the agent can leverage established structural priors to produce levels that are not only valid but also more varied, more natural, and more similar to classically-generated content. This shows that previous PCG methods are not made obsolete by LLMs. Instead, they become modular components that an LLM agent can invoke as part of a unified editing and optimization process.

References

[1] N. Shaker, J. Togelius, and M. J. Nelson, "Procedural content generation in games," 2016.

[2] A. Khalifa, R. Gallotta, M. Barthet, A. Liapis, J. Togelius, and G. N. Yannakakis, "The procedural content generation benchmark: An open-source testbed for generative challenges in games," in *Proceedings of the 20th International Conference on the Foundations of Digital Games*, ser. FDG '25. New York, NY, USA: Association for Computing Machinery, 2025. [Online]. Available: https://doi.org/10.1145/3723498.3723794

[3] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[4] R. Gallotta, K. Arulkumaran, and L. B. Soros, "Evolving spaceships with a hybrid l-system constrained optimisation evolutionary algorithm," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2022, pp. 711–714.

[5] A. Khalifa, M. C. Green, G. Barros, and J. Togelius, "Intentional computational level design," in *Proceedings of The Genetic and Evolutionary Computation Conference*, 2019, pp. 796–803.

[6] N. Shaker, M. Nicolau, G. N. Yannakakis, J. Togelius, and M. O'neill, "Evolving levels for super mario bros using grammatical evolution," in *2012 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2012, pp. 304–311.

[7] M. Cook, S. Colton, A. Raad, and J. Gow, "Mechanic miner: Reflection-driven game mechanic discovery and level design," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2013, pp. 284–293.

[8] C. B. Browne, "Automatic generation and evaluation of recombination games," Ph.D. dissertation, Queensland University of Technology, 2008.

[9] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius, "Procedural content generation via machine learning (pcgml)," *IEEE Transactions on Games*, vol. 10, no. 3, pp. 257–270, 2018.

[10] S. Dahlskog, J. Togelius, and M. J. Nelson, "Linear levels through n-grams," in *Proceedings of the 18th International Academic MindTrek Conference: Media Business, Management, Content & Services*, 2014, pp. 200–206.

[11] S. Snodgrass and S. Ontanón, "Learning to generate video game maps using markov models," *IEEE transactions on computational intelligence and AI in games*, vol. 9, no. 4, pp. 410–422, 2016.

[12] A. Summerville and M. Mateas, "Super mario as a string: Platformer level generation via lstms," *arXiv preprint arXiv:1603.00930*, 2016.

[13] A. Sarkar, Z. Yang, and S. Cooper, "Conditional level generation and game blending," *arXiv preprint arXiv:2010.07735*, 2020.

[14] V. Volz, J. Schrum, J. Liu, S. M. Lucas, A. Smith, and S. Risi, "Evolving mario levels in the latent space of a deep convolutional generative adversarial network," in *Proceedings of the genetic and evolutionary computation conference*, 2018, pp. 221–228.

[15] G. Todd, S. Earle, M. U. Nasir, M. C. Green, and J. Togelius, "Level generation through large language models," in *Proceedings of the 18th International Conference on the Foundations of Digital Games*, 2023, pp. 1–8.

[16] S. Sudhakaran, M. González-Duque, M. Freiberger, C. Glanois, E. Najarro, and S. Risi, "Mariogpt: Open-ended text2level generation through large language models," *Advances in Neural Information Processing Systems*, vol. 36, pp. 54 213–54 227, 2023.

[17] T. Merino, S. Earle, R. Sudhakaran, S. Sudhakaran, and J. Togelius, "Making new connections: Llms as puzzle generators for the new york times' connections word game," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 20, no. 1, 2024, pp. 87–96.

[18] S. Huang, "Word2minecraft: Generating 3d game levels through large language models," Master's thesis, New York University Tandon School of Engineering, 2025.

[19] M. U. Nasir, Y. Li, S. James, and J. Togelius, "Mortar: Evolving mechanics for automatic game design," *arXiv preprint arXiv:2601.00105*, 2025.

[20] S. Earle, A. Khalifa, M. U. Nasir, Z. Jiang, G. Todd, A. Banburski-Fahey, and J. Togelius, "Scriptdoctor: Automatic generation of puzzlescript games via large language models and tree search," in *2025 IEEE Conference on Games (CoG)*. IEEE, 2025, pp. 1–5.

[21] R. Gallotta, A. Liapis, and G. Yannakakis, "Llmaker: A game level design interface using (only) natural language," in *2024 IEEE Conference on Games (CoG)*. IEEE, 2024, pp. 1–2.

[22] R. Gallotta, A. Liapis, and G. N. Yannakakis, "Freyr: A framework for recognizing and executing your requests," *arXiv preprint arXiv:2501.12423*, 2025.

[23] A. Liapis, G. Smith, and N. Shaker, "Mixed-initiative content creation," in *Procedural content generation in games*. Springer, 2016, pp. 195–214.

[24] A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, "Pcgrl: Procedural content generation via reinforcement learning," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 16, no. 1, 2020, pp. 95–101.

[25] S. Earle, M. Edwards, A. Khalifa, P. Bontrager, and J. Togelius, "Learning controllable content generators," in *2021 IEEE Conference on Games (CoG)*, 2021, pp. 1–9.

[26] Z. Jiang, S. Earle, M. Green, and J. Togelius, "Learning controllable 3d level generators," in *Proceedings of the 17th International Conference on the Foundations of Digital Games*, ser. FDG '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3555858.3563273

[27] S. Earle, Z. Jiang, and J. Togelius, "Scaling, control and generalization in reinforcement learning level generators," in *2024 IEEE Conference on Games (CoG)*, 2024, pp. 1–8.

[28] S. Earle, Z. Jiang, E. Vinitsky, and J. Togelius, "Video game level design as a multi-agent reinforcement learning problem," *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 21, no. 1, pp. 32–42, Nov. 2025. [Online]. Available: https://ojs.aaai.org/index.php/AIIDE/article/view/36807

[29] T. T. Shigeru Miyamoto, "The legend of zelda," Nintendo Entertainment System, 1986, video game.

[30] D. Smith, "Lode runner," Apple II, 1983, pC.

[31] T. Rabbit, "Sokoban," PC, 1982.

[32] T. T. Shigeru Miyamoto, "Super mario bros." Nintendo Entertainment System, 1985, video game.

[33] Anthropic, "Claude sonnet 4.6 system card," https://www-cdn.anthropic.com/bbd8ef16d70b7a1665f14f306ee88b53f686aa75.pdf, Feb. 2026, accessed: 2026-03-24.

[34] ——, "Claude opus 4.6 system card," https://www-cdn.anthropic.com/6a5fa276ac68b9aeb0c8b6af5fa36326e0e166dd.pdf, Feb. 2026, accessed: 2026-03-24.

[35] G. Comanici, E. Bieber, M. Schaekermann, I. Pasupat, N. Sachdeva, I. Dhillon, M. Blistein, O. Ram, D. Zhang, E. Rosen *et al.*, "Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities," *arXiv preprint arXiv:2507.06261*, 2025.

[36] Google DeepMind, "Gemini 3 flash model card," https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Flash-Model-Card.pdf, Dec. 2025, published: December 2025. Accessed: 2026-03-24.

[37] ——, "Gemini 3 pro model card," https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Pro-Model-Card.pdf, Dec. 2025, model card update: December 2025. Accessed: 2026-03-24.

[38] OpenAI, "Gpt-4o system card," https://cdn.openai.com/gpt-4o-system-card.pdf, Aug. 2024, accessed: 2026-03-24.

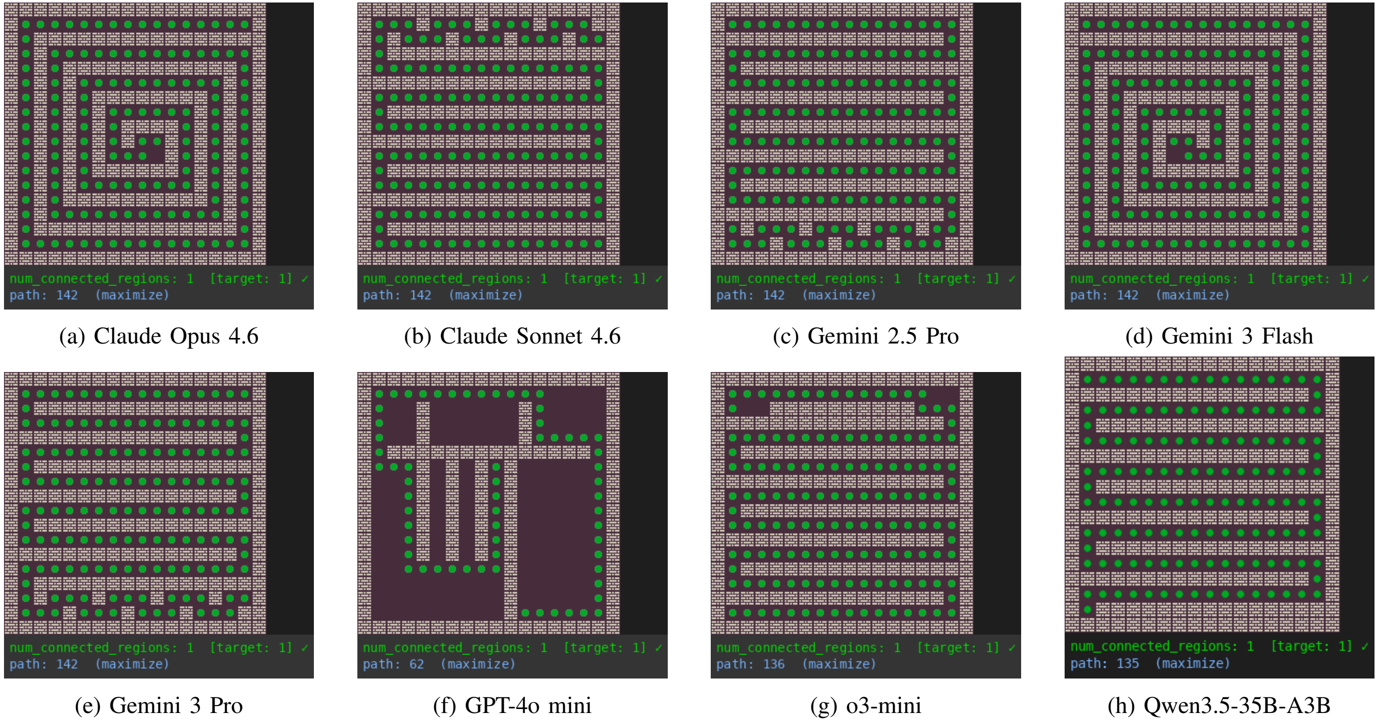
[39] ——, "Openai o3-mini system card," https://cdn.openai.com/o3-mini-system-card-feb10.pdf, Jan. 2025, accessed: 2026-03-24.

[40] Qwen Team, "Qwen3.5: Towards native multimodal agents," February 2026. [Online]. Available: https://qwen.ai/blog?id=qwen3.5

[41] K. Ellis, C. Wong, M. Nye, M. Sablé-Meyer, L. Morales, L. Hewitt, L. Cary, A. Solar-Lezama, and J. B. Tenenbaum, "Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning," in *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation*, 2021, pp. 835–850.

APPENDIX
MORE QUALITATIVE RESULTS

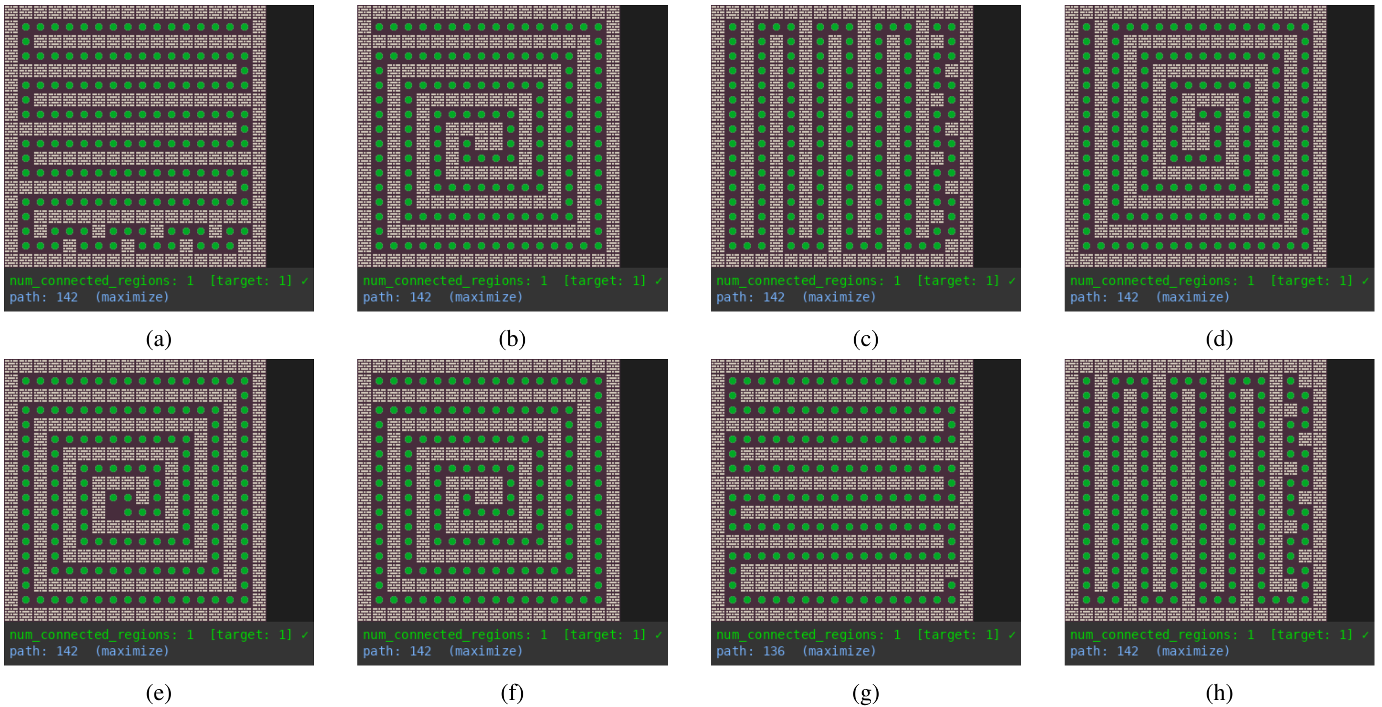
Binary



(a) Claude Opus 4.6 (b) Claude Sonnet 4.6 (c) Gemini 2.5 Pro (d) Gemini 3 Flash

(e) Gemini 3 Pro (f) GPT-4o mini (g) o3-mini (h) Qwen3.5-35B-A3B

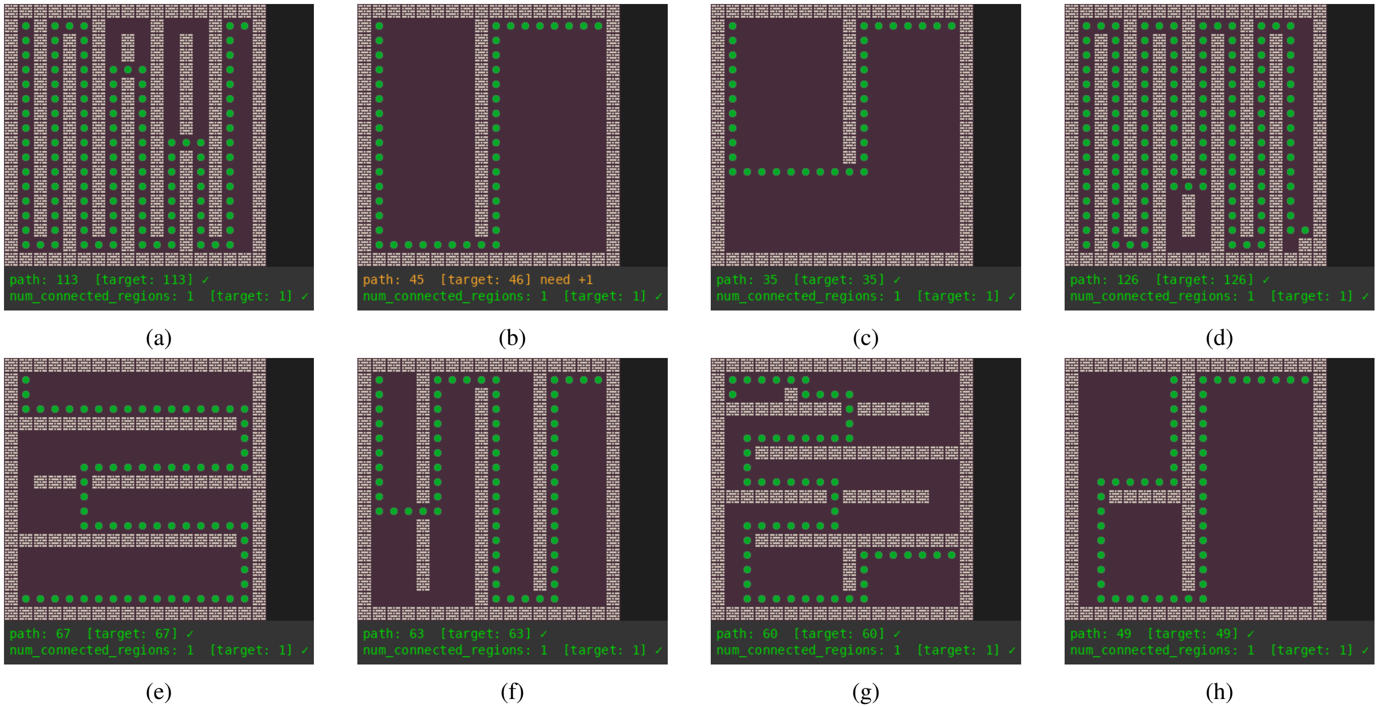
Fig. 8: Qualitative comparison of levels generated by different backbone models in the binary problem experiment under the same target metrics.



(a) (b) (c) (d)

(e) (f) (g) (h)

Fig. 9: Qualitative diversity of levels generated by Gemini 3 Pro in the binary problem experiment under the same target metrics.

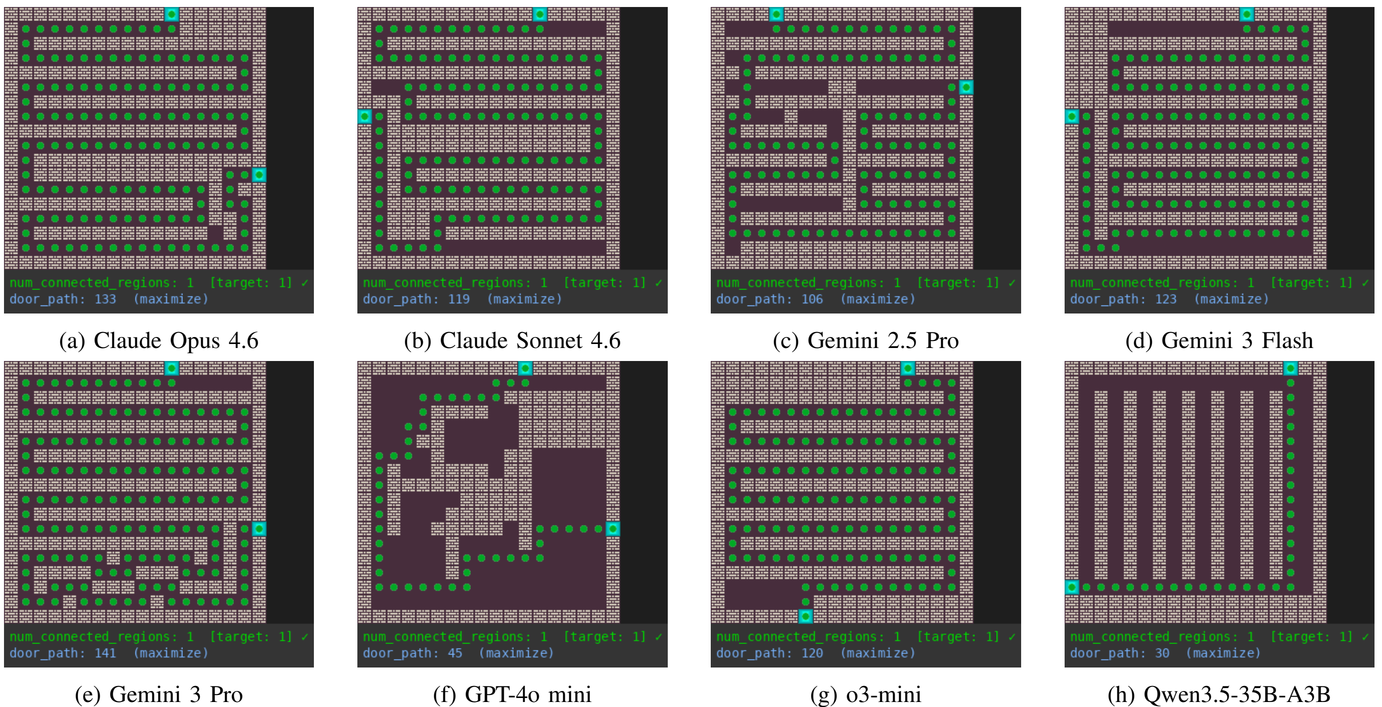


(a) (b) (c) (d)

(e) (f) (g) (h)

Fig. 10: Qualitative controllability examples generated by Gemini 2.5 Pro in the binary problem experiment under different target metrics.

Binary Door



(a) Claude Opus 4.6 (b) Claude Sonnet 4.6 (c) Gemini 2.5 Pro (d) Gemini 3 Flash

(e) Gemini 3 Pro (f) GPT-4o mini (g) o3-mini (h) Qwen3.5-35B-A3B

Fig. 11: Qualitative comparison of levels generated by different backbone models in the binary door problem experiment under the same target metrics.

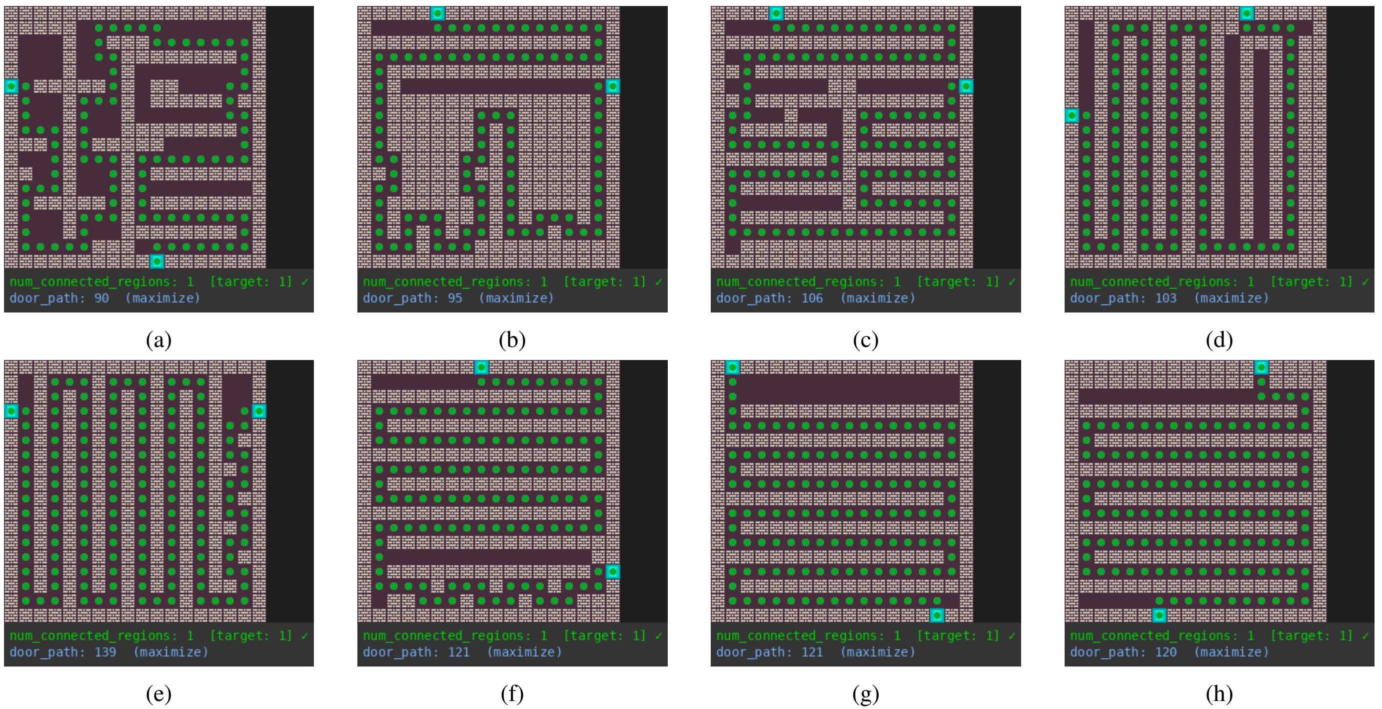


Fig. 12: Qualitative diversity of levels generated by Gemini 2.5 Pro in the binary door problem experiment under the same target metrics.

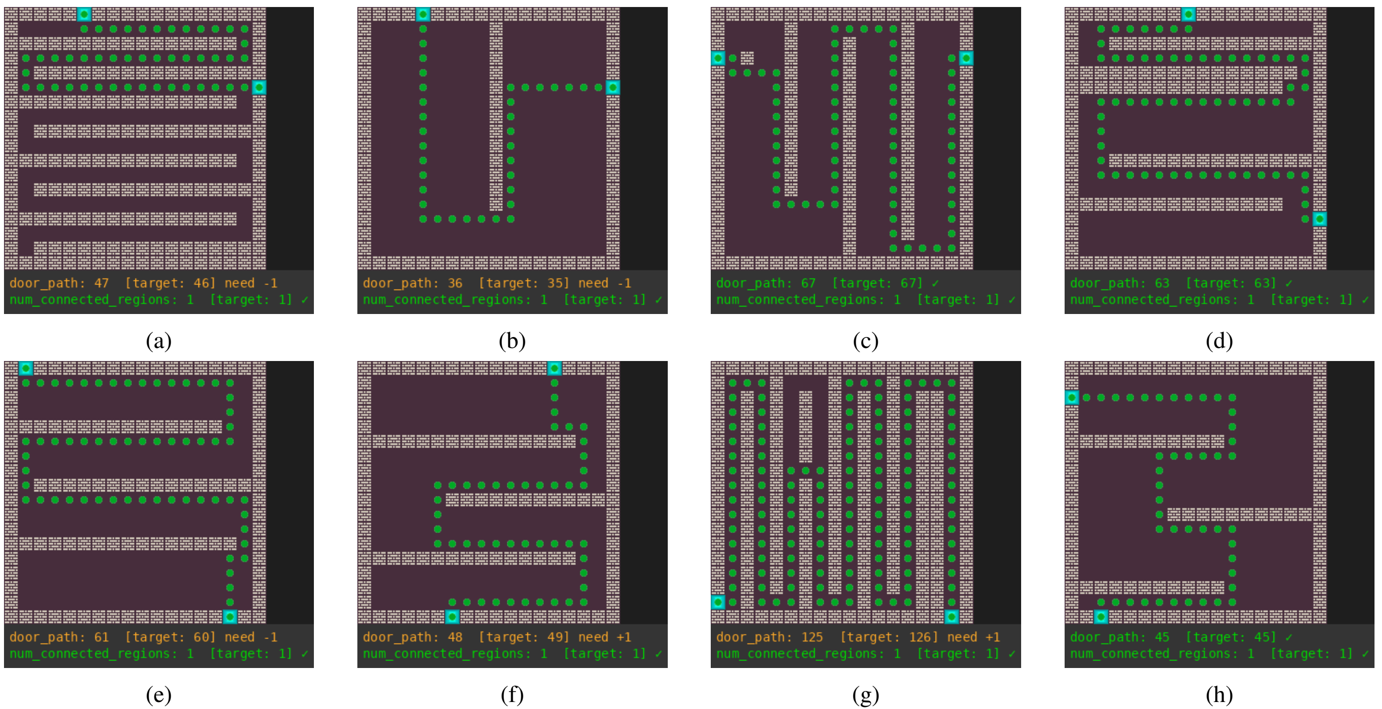


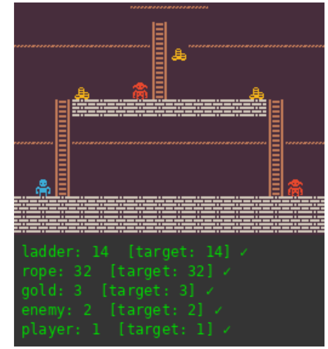
Fig. 13: Qualitative controllability examples generated by Gemini 2.5 Pro in the binary door problem experiment under different target metrics.

Lode Runner

(a) Claude Opus 4.6

(b) Claude Sonnet 4.6

(c) Gemini 2.5 Pro



(d) Gemini 3 Flash

(e) Gemini 3 Pro

(f) GPT-4o mini

(g) o3-mini

(h) Qwen3.5-35B-A3B

Fig. 14: Qualitative comparison of levels generated by different backbone models in the Lode Runner problem experiment under the same target metrics.

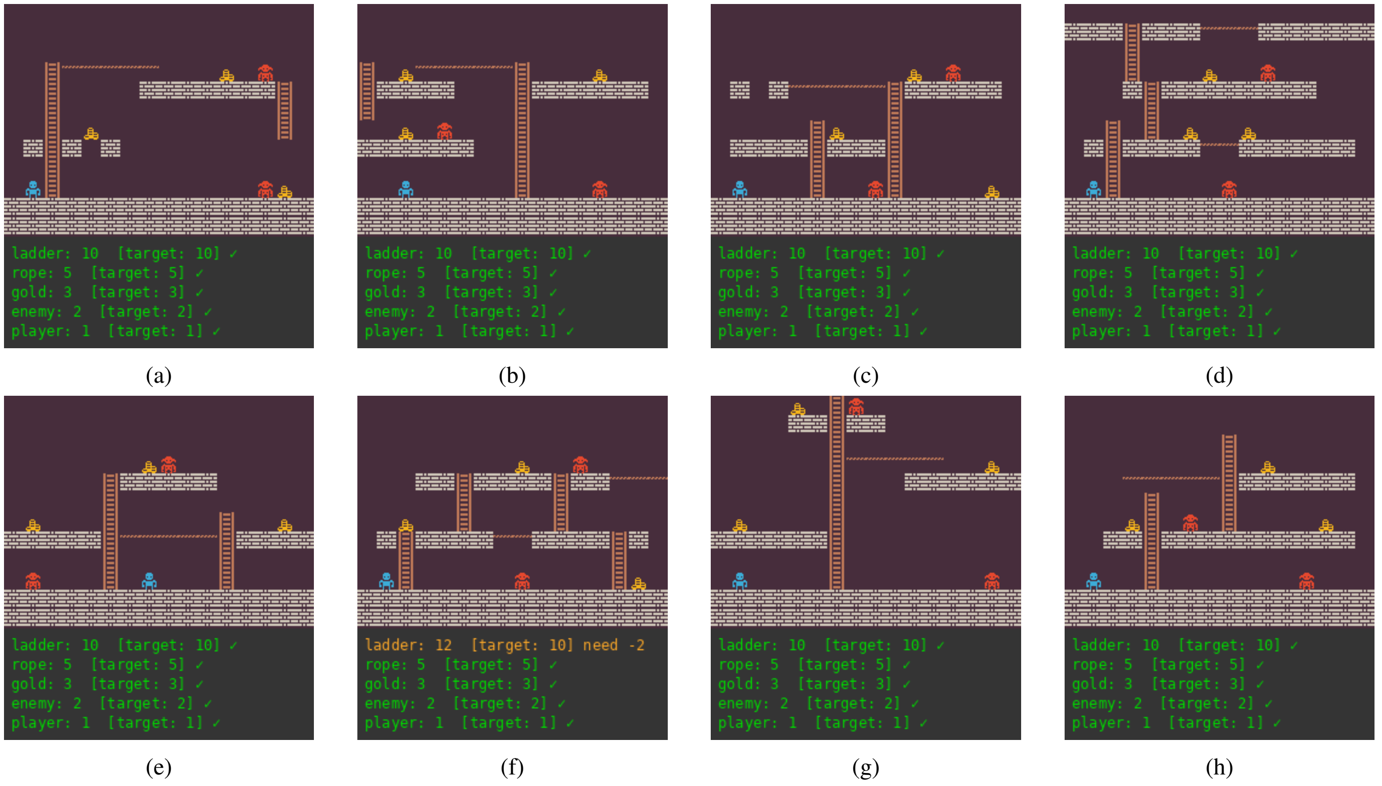


Fig. 15: Qualitative diversity of levels generated by Gemini 3 Pro in the Lode Runner problem experiment under the same target metrics.

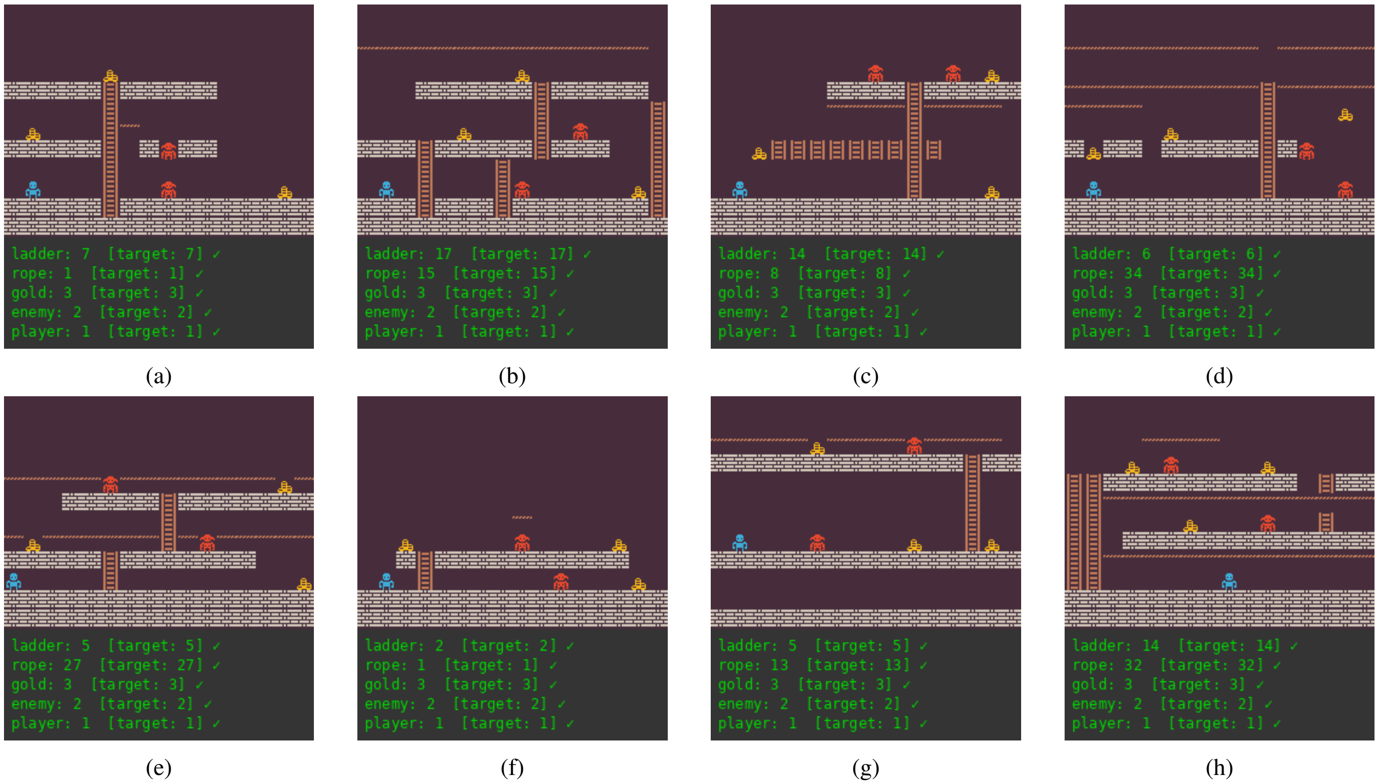
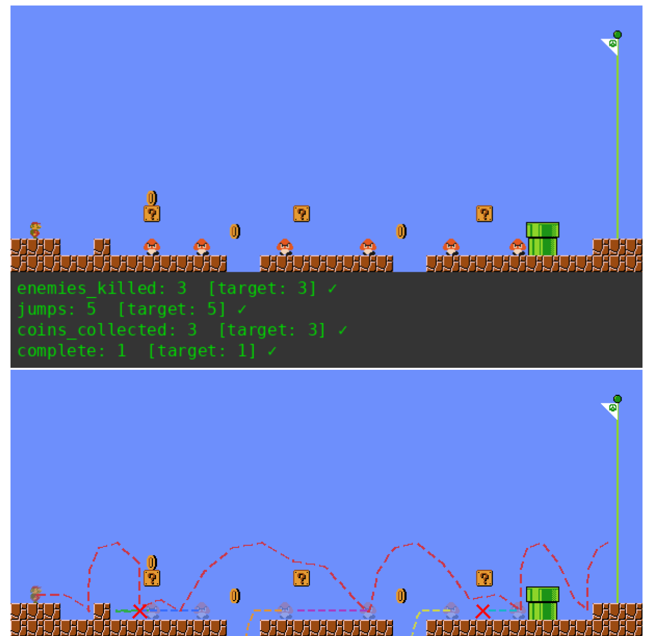


Fig. 16: Qualitative controllability examples generated by Gemini 2.5 Pro in the Lode Runner problem experiment under different target metrics.

(a) Claude Opus 4.6

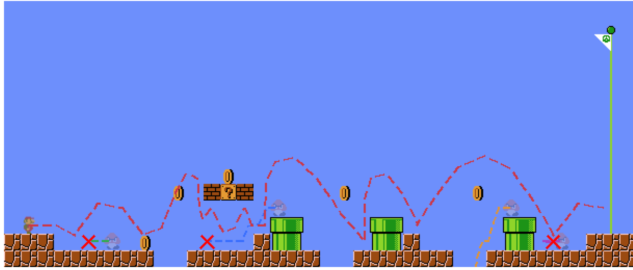
(b) Claude Sonnet 4.6

(c) Gemini 2.5 Pro

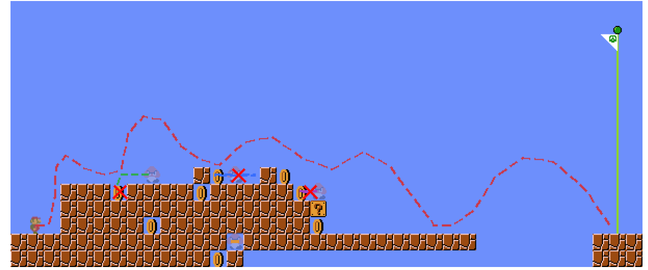


(d) Gemini 3 Flash

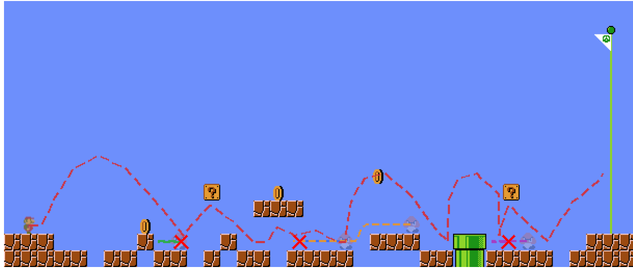
Fig. 17: Qualitative comparison of levels generated by different backbone models in the SMB (A*) problem experiment under the same target metrics (1/2). Top: level layout. Bottom: agent trajectory.



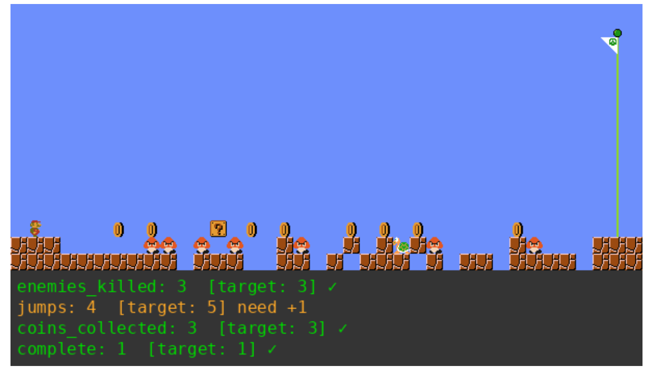
(a) Gemini 3 Pro

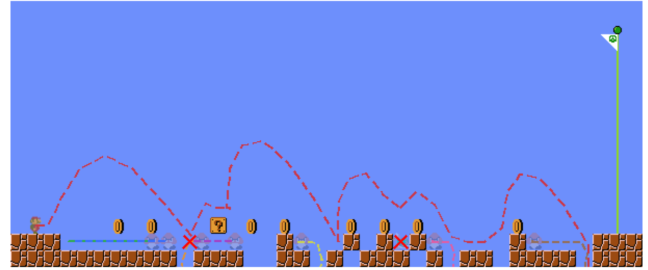


(b) GPT-4o mini



(c) o3-mini





(d) Qwen3.5-35B-A3B

Fig. 18: Qualitative comparison of levels generated by different backbone models in the SMB (A*) problem experiment under the same target metrics (2/2).

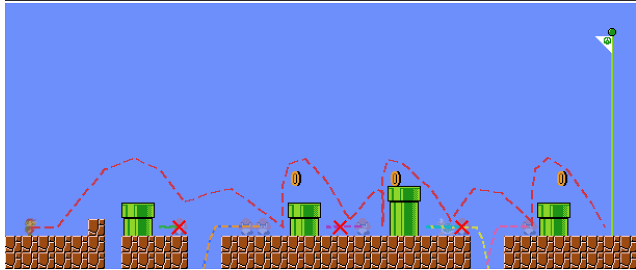
(a)

(b)

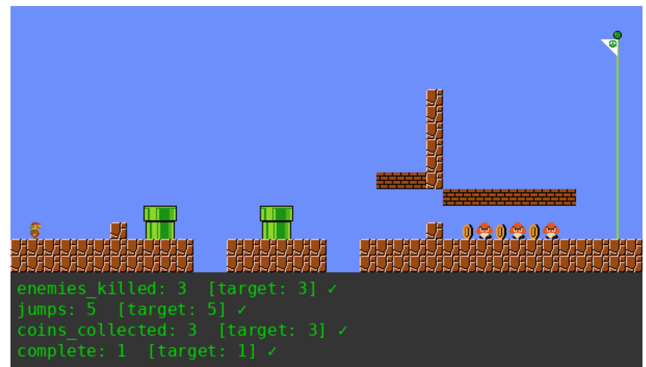
(c)

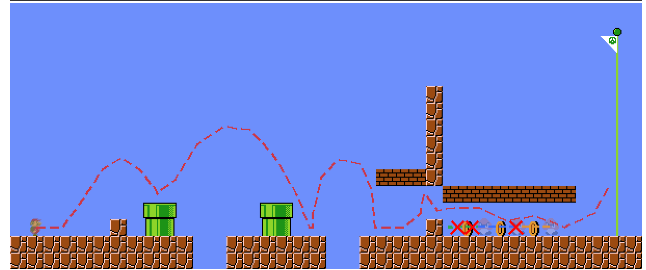
(d)

Fig. 19: Qualitative diversity of levels generated by Gemini 3 Pro in the SMB (A*) problem experiment (1/2). Top: level layout. Bottom: agent trajectory.

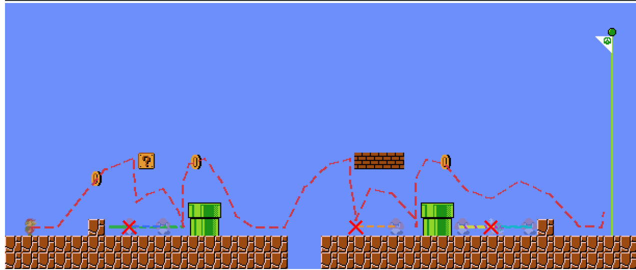


(a)

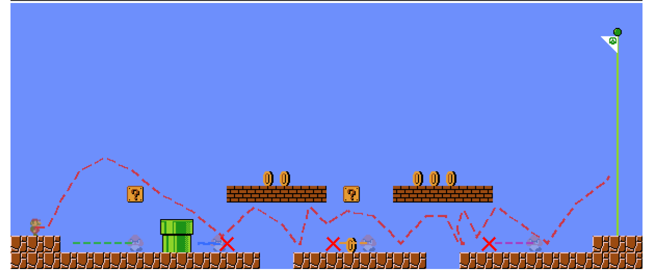




(b)



(c)



(d)

Fig. 20: Qualitative diversity of levels generated by Gemini 3 Pro in the SMB (A*) problem experiment (2/2).

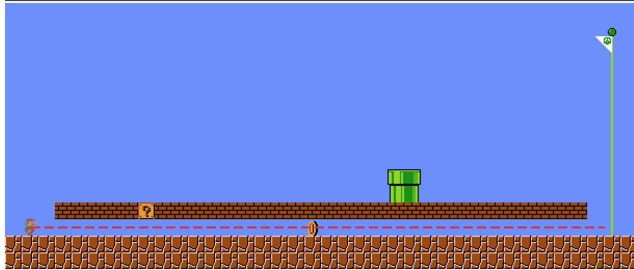
(a)

(b)

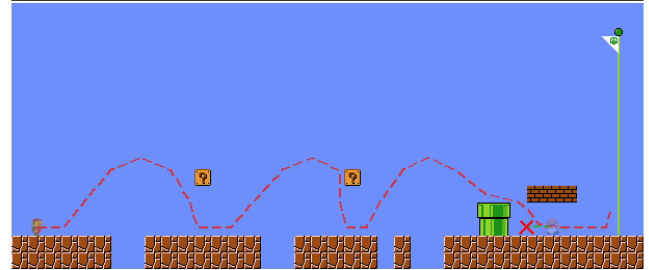
(c)

(d)

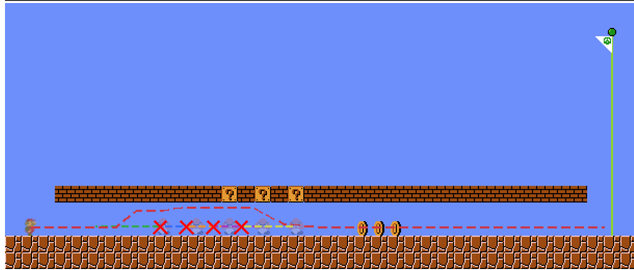
Fig. 21: Qualitative controllability examples generated by Gemini 3 Flash in the SMB (A*) problem experiment under different target metrics (1/2). Top: level layout. Bottom: agent trajectory.



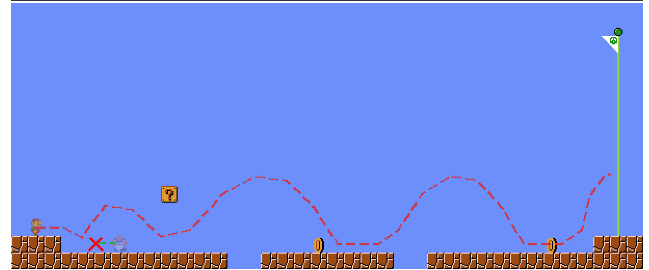
(a)



(b)

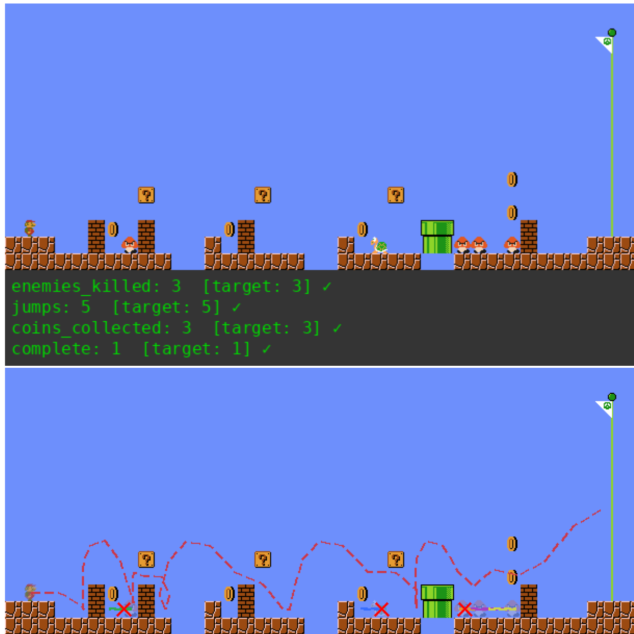


(c)



(d)

Fig. 22: Qualitative controllability examples generated by Gemini 3 Flash in the SMB (A*) problem experiment under different target metrics (2/2).



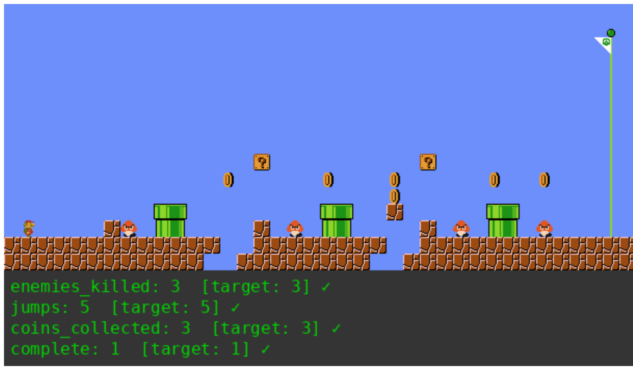
(a) Claude Opus 4.6

(b) Claude Sonnet 4.6

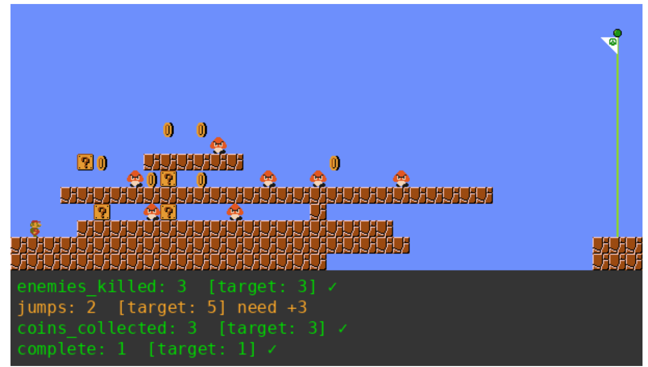
(c) Gemini 2.5 Pro

(d) Gemini 3 Flash

Fig. 23: Qualitative comparison of levels generated by different backbone models in the SMB (Auto) problem experiment under the same target metrics (1/2). Top: level layout. Bottom: agent trajectory.



(a) Gemini 3 Pro



(b) GPT-4o mini

(c) o3-mini

(d) Qwen3.5-35B-A3B

Fig. 24: Qualitative comparison of levels generated by different backbone models in the SMB (Auto) problem experiment under the same target metrics (2/2).

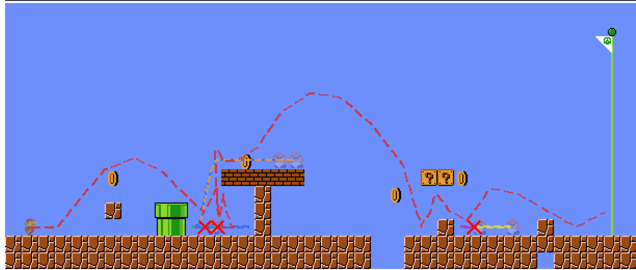
(a)

(b)

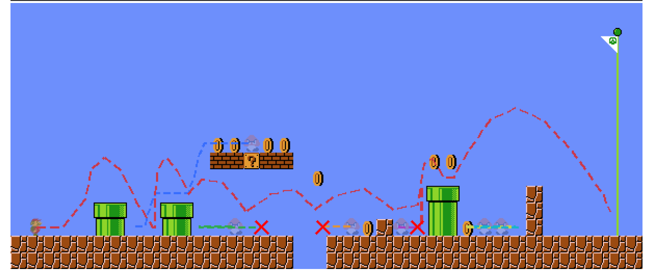
(c)

(d)

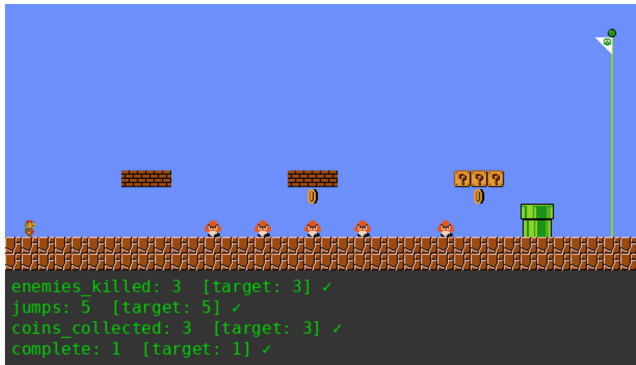
Fig. 25: Qualitative diversity of levels generated by Gemini 3 Pro in the SMB (Auto) problem experiment (1/2). Top: level layout. Bottom: agent trajectory.

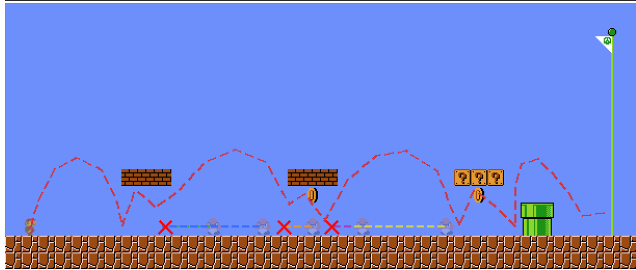


(a)

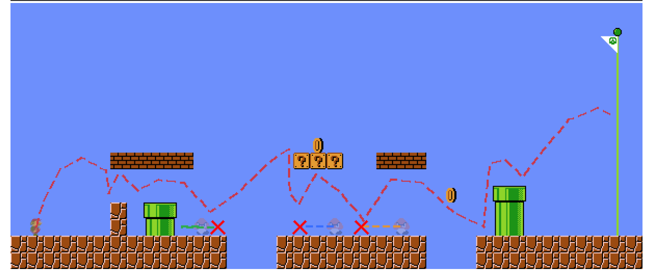


(b)



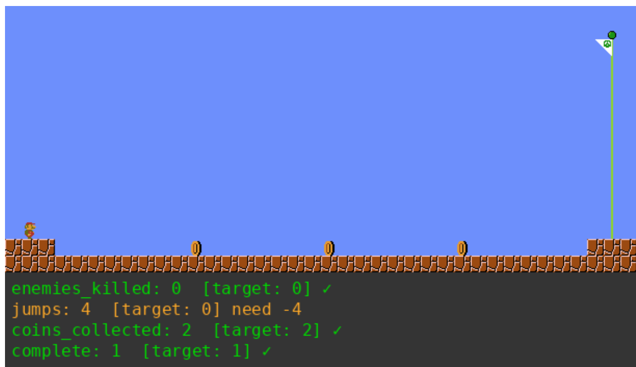


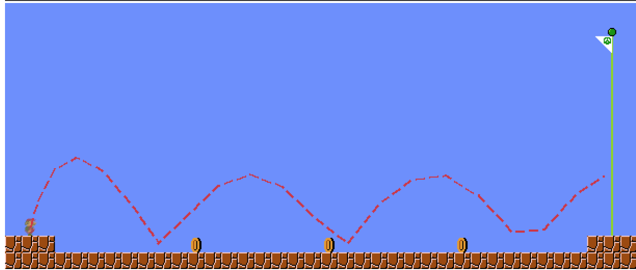
(c)



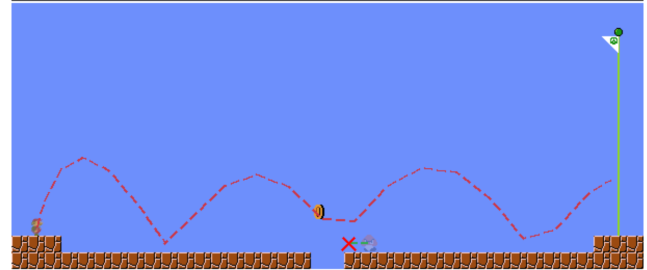
(d)

Fig. 26: Qualitative diversity of levels generated by Gemini 3 Pro in the SMB (Auto) problem experiment (2/2).

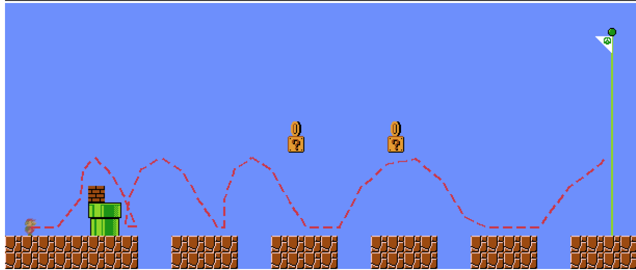




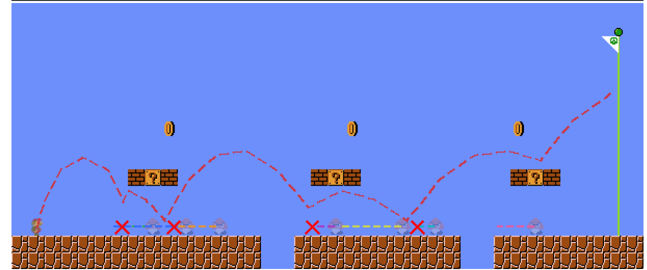
(a)



(b)



(c)



(d)

Fig. 27: Qualitative controllability examples generated by Gemini 3 Flash in the SMB (Auto) problem experiment under different target metrics (1/2). Top: level layout. Bottom: agent trajectory.

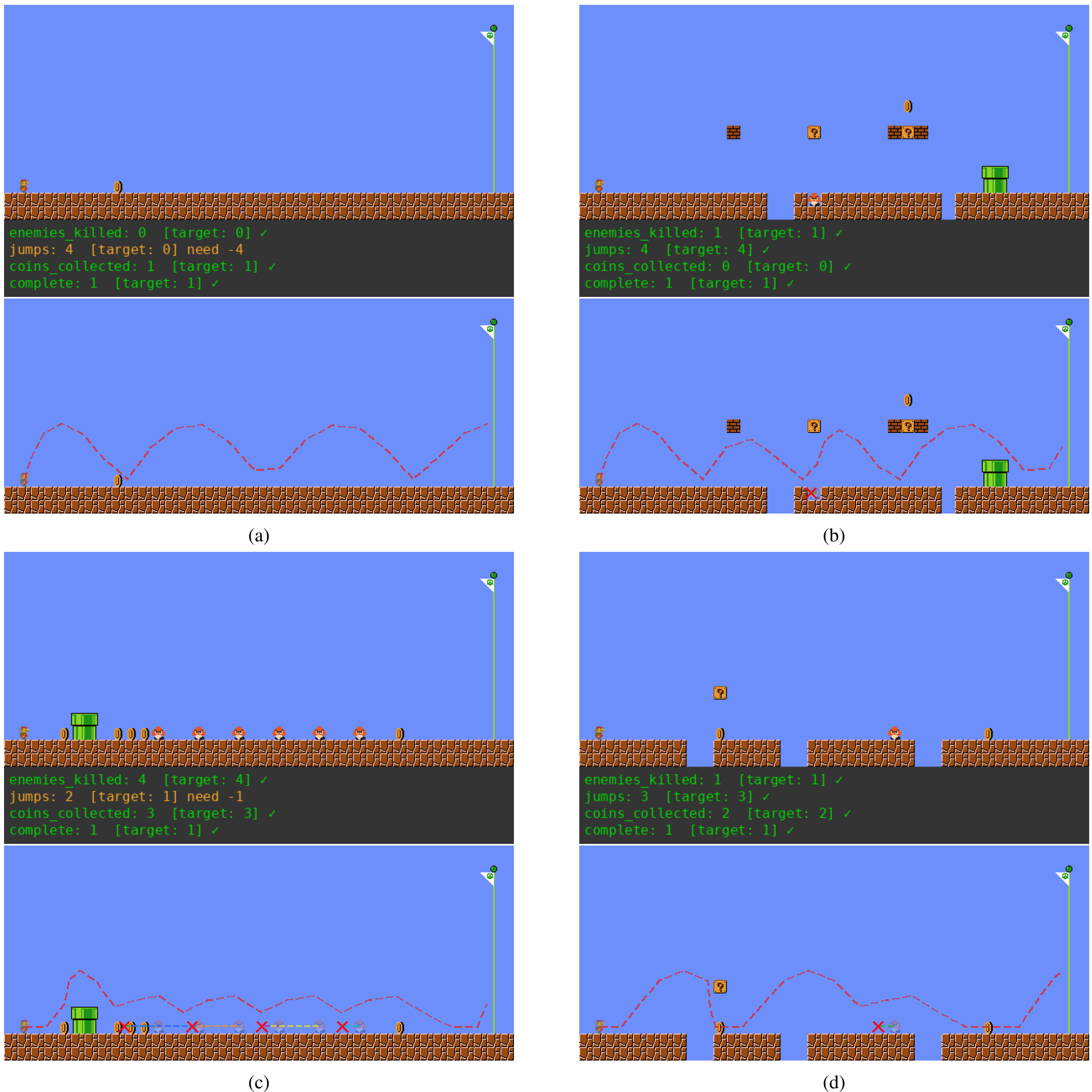


Fig. 28: Qualitative controllability examples generated by Gemini 3 Flash in the SMB (Auto) problem experiment under different target metrics (2/2).

Environment Details

A. Binary Maze

The binary maze problem requires the agent to produce a two-dimensional grid of size $16(height) \times 16(width)$, containing exactly two tile types: empty and wall. An agent may traverse empty tiles using four-directional movement (up, down, left, and right). For this problem, we focus on 2 main metrics: longest shortest path between any two pairs of traversable points, and numbers of connected region. We calculate the longest shortest path and numbers of regions using Dijkstra algorithm and flood filling algorithm, respectively, the process is same as [27].

- Quality: For quality metric in pcg benchmark, a well-formed maze should contain a single connected component of empty tiles, ensuring that every passable cell is reachable from every other. Also the longest shortest path should exceed half

Algorithm 1 Random Door Placement on the Augmented Grid Perimeter. Design rationale: seeded per `door_seed` for reproducibility; enforce minimum perimeter separation $\texttt{peri_dist} \geq \min(W, H)$ to avoid trivial adjacent doors.

Require: Augmented grid width W, height H; RNG seed s (door_seed, default 42)
Ensure: Two door coordinates $\texttt{door1}, \texttt{door2}$ on non-corner perimeter cells

1: **Build perimeter cycle P (clockwise, excluding corners).**
2: $P \leftarrow [\,]$
3: Append top row left→right (excluding corners) to P
4: Append right column top→bottom (excluding corners) to P
5: Append bottom row right→left (excluding corners) to P
6: Append left column bottom→top (excluding corners) to P
7: $L \leftarrow |P|$

8: **Pick the first door (seeded).**
9: Initialize RNG with seed s
10: $\texttt{idx1} \leftarrow \texttt{randint}(0, L-1)$
11: $\texttt{door1} \leftarrow P[\texttt{idx1}]$

12: **Find valid candidates for the second door.**
13: $d_{\min} \leftarrow \min(W, H)$
14: $C \leftarrow [\,]$
15: **for** $i = 0$ to $L-1$ **do**
16:     **if** $i = \texttt{idx1}$ **then**
17:         **continue**
18:     **end if**
19:     $d_{\text{fwd}} \leftarrow (i - \texttt{idx1}) \bmod L$
20:     $d_{\text{bwd}} \leftarrow (\texttt{idx1} - i) \bmod L$
21:     $\texttt{peri_dist} \leftarrow \min(d_{\text{fwd}}, d_{\text{bwd}})$
22:     **if** $\texttt{peri_dist} \geq d_{\min}$ **then**
23:         Append i to C
24:     **end if**
25: **end for**

26: **Pick the second door (with fallback).**
27: **if** $|C| > 0$ **then**
28:     $\texttt{idx2} \leftarrow \texttt{choice}(C)$
29: **else**
30:     $\texttt{idx2} \leftarrow \arg\max_{i \in \{0,\ldots,L-1\} \setminus \{\texttt{idx1}\}} \min\big((i - \texttt{idx1}) \bmod L, (\texttt{idx1} - i) \bmod L\big)$
31: **end if**
32: $\texttt{door2} \leftarrow P[\texttt{idx2}]$
33: **return** $(\texttt{door1}, \texttt{door2})$

of the maximum possible path length. To simply the calculation process, we use the length of pure zig-zag that fill the whole map as a approximate of the maximum possible path in the map.

- Controllability: the controllability score of binary centers the plateau on the sampled control value between add/subtract cerror. In this problem we only measure the controllability of the path length and the acceptable error is 10% of the sampled target.
- Diversity: the diversity score for binary maze measures Hamming distance of all the grids between two levels.

B. Binary Maze with Doors

For the binary maze setting, maximizing the longest possible path tends to produce highly degenerate solutions. On a fixed-size grid, the optimal construction is essentially limited to a small set of near-perfect zigzag patterns, resulting in low structural diversity. To alleviate this issue, we introduce a binary-with-doors variant, inspired by prior work [26]: two boundary doors (entrance/exit cells) are sampled randomly according to a predefined procedure 1. Compared to the vanilla binary setting, the LLM agent not only needs to construct a maze that meets the target metrics, but also ensure that the two sampled doors are

connected. Because the door locations are random, the true longest path connecting an arbitrary pair of doors is closely related to finding a Hamiltonian path, which is NP-complete in general. Consequently, in our metric computation we approximate the maximum achievable path length using the length of the longest zigzag pattern on the same board size, and use this value to normalize/score path-length performance.

- Quality: The quality score is the arithmetic mean of two trapezoidal reward components:
- Controllability: The controllability score again measures how precisely the agent can steer connected path to a specified target value. Only solvable trials (where connected path > 0) contribute to the aggregate controllability score; unsolvable trials receive a score of 0
- Diversity: Diversity between two levels is measured via normalized Hamming distance over their flattened binary representations. This is computed pairwise across all solvable trials (where connected path > 0) that have connected path in a diversity experiment.

C. Zelda

The Zelda dungeon problem requires the agent to produce a top-down two-dimensional grid based game of size $16(height) \times 16(width)$, containing six tile types: wall, empty, player, key, door, and enemy. The player needs to navigate to collect a key, then reach an exit door, while killing the enemies in the way. For this problem, we focus on 4 main metrics: shortest path length from player to key (`player key`), shortest path length from key to door (`key door`), number of connected regions, and entity counts (players, keys, doors, enemies). We calculate the shortest paths using Dijkstra's algorithm, treating enemy tiles as traversable tiles. The number of connected regions is computed via flood fill.

- Quality: A well-formed Zelda level must contain exactly one player, one key, and one door, with a single connected region of passable tiles. The level must be playable, meaning a path exists from the player to the key and from the key to the door. Additionally, the combined solution length (`player key` + `key door`) should be close to a target length of $W + H$ (32 for a 16×16 level), and the enemy count should be close to its target value. The quality score averages four sub-scores: region correctness, entity count accuracy, and playability (which itself includes a solution length reward when both paths exist).
- Controllability: The controllability score measures how precisely the agent can achieve sampled target values for `player key` and `key door` independently. The acceptable error is 10% of each sampled target value. The final controllability score is the average of the two path length scores. Only solvable levels (both paths exist) contribute.
- Diversity: The diversity score compares the solution paths of two levels. Each level's path is the concatenation of the player-to-key and key-to-door coordinate sequences. Before comparison, paths are normalized by mirroring coordinates when the starting point falls in the right or bottom half of the grid to account for spatial symmetry. The similarity between two normalized paths is computed using Python's `SequenceMatcher`, which finds the longest common subsequences and returns a ratio in $[0, 1]$. The diversity score reaches 1.0 when the path dissimilarity ($1 - \text{ratio}$) exceeds 0.3, and ramps linearly from 0 to 1 for smaller differences.

D. Lode runner

Lode Runner is a two dimensional $11(\text{height}) \times 16(\text{width})$ side-view platformer level design task with seven tile types: solid, empty, player, gold, enemy, ladder, and rope. The player may walk on supported empty tiles, climb ladders, move along ropes, and fall when unsupported. Rather than simulating gameplay in real time, we evaluate levels with a static BFS reachability analysis over states $(x, y, \text{falling})$ starting from the player position. The resulting reachability map determines whether all gold is collectable and how much of the structural layout is traversed. Because gold and enemies are treated as static markers, the problem reduces to constrained reachability. A valid playable level must contain exactly one player, at least one gold tile, and permit collection of all gold. For this problem, we focus on two controllable metrics: the number of ladder tiles and the number of rope tiles.

- Quality: The PCG benchmark evaluates quality through four progressive tiers, each gated on the previous tier achieving a score of ≥ 1.0. Tier 1 (Stats) checks that entity counts are reasonable: exactly 1 player, gold count near a target, and enemy count near a target. Tier 2 (Exploration) requires that at least 20% of the level is reachable by the player via BFS exploration. Tier 3 (Playability) requires that all gold is collectible and that at least 75% of structural tiles (ladders, ropes, floor supports) are actually traversed by the player. Tier 4 (Decoration) measures how well the level's movement pattern distributions (walking, hanging, climbing, falling) match reference levels from the original game using Jensen–Shannon divergence, and penalizes excessive disconnected structural regions. The final quality score is the average of all four tiers, each in $[0, 1]$.
- Controllability: The controllability score measures how closely the generated level matches sampled target counts for ladder and rope tiles. Each metric is scored again using a trapezoidal reward function centered on the target value with an acceptable error of 2% of the total grid area. The final controllability score is the average of the ladder and rope scores.

Only playable levels (all gold reachable by the player) contribute to the controllability score and unplayable levels receive a score of 0.

- Diversity: The diversity score compares the exploration maps of two levels by computing the element-wise difference across four movement-type layers (walking, climbing, hanging, and falling). These differences are combined with weights of 0.3 each for walking, climbing, and hanging, and 0.1 for falling. The weighted sum is scored against a target of 40% of the total grid area differing between the two levels. Only playable levels are included in the diversity computation.

E. Sokoban

The Sokoban problem requires the agent to produce a two-dimensional grid based puzzle game level of size $8(\text{height}) \times 8(\text{width})$, containing five tile types: solid (wall), empty (floor), player, crate, and target. The player can move in four directions (up, down, left, and right). When the player moves into a crate, the crate is pushed one tile in that direction, and crates cannot be pushed through walls or other crates or be pulled. The puzzle is solved when all crates are placed on target locations. A valid level must contain exactly one player and an equal number of crates and targets. For this problem, the main metrics are: the number of crates, solvability status, and solution length (number of moves to solve).

Solvability is determined by a cascade of four solvers, each limited to 5,000 node expansions: BFS first, then A* with decreasing cost weights (balance $\in \{1.0, 0.5, 0.0\}$), where the priority function is $f = h + \text{balance} \cdot g$, where h is the sum of Manhattan distances from each crate to its nearest unmatched target (greedy one-to-one assignment), and g is the search depth. A level is solvable if any of the four solvers reaches a winning state within its budget. During search, states where a crate is pushed into a deadlock position (e.g., a corner not on a target) are pruned. Different balance values explore the search space differently: BFS is optimal but exhausts its budget quickly on complex puzzles, while pure greedy (balance $= 0$) aggressively chases low-heuristic states and can find solutions that BFS misses.

- Quality: The quality metric combines two components averaged equally: (1) a structural score, and (2) a solver score. The structural score is the mean of three sub-rewards that each range from 0 to 1: player count (peaks at exactly 1), crate count (peaks at 1 or more), and crate-target balance (peaks when the difference is 0). Each sub-reward increases linearly from 0 at the boundary to 1 at the target plateau. The solver score is only computed when the solver successfully runs (i.e., the level has valid structure); otherwise it is 0. It is the mean of two sub-rewards: a heuristic reward that peaks when $h = 0$ (all crates on targets) and decays linearly as h increases, and a solution length reward that peaks when the solution is at least $(W + H) \times$ difficulty moves long, where difficulty is a per-variant constant set to 1 for the standard 8×8 variant.
- Controllability: The controllability score of Sokoban measures how closely the generated level matches a sampled target crate count. The reward plateaus at full score when the actual crate count is within ±1 of the target, and decreases linearly outside that range. Only solvable levels (solution length > 0) contribute to the controllability score; unsolvable levels receive a score of 0.
- Diversity: The diversity score for Sokoban measures pairwise dissimilarity of solutions between levels. Each solution is first canonicalized (flipped to a consistent orientation) and then compressed by collapsing consecutive identical moves into a single character (e.g., "rrruuuddd" → "rud"). Pairwise similarity is computed using `SequenceMatcher`, and the diversity reward requires at least 50% difference for full score. Only solvable levels are included in pairwise comparisons.

F. Super Mario Bros

The Super Mario Bros (SMB) problem requires the agent to produce a two-dimensional grid of side-view platformer game of size $16(\text{height}) \times 32(\text{width})$, containing ten tile types: empty, solid, ladder, brick, question block, tube, coin, goomba, koopa, and spiny. Unlike other problems where metrics are computed from the level layout alone, SMB metrics are gameplay simulation outcomes obtained by running a Mario AI agent through a physics simulation of the generated level. The simulation uses a pure-Python reimplementation of the Mario AI Framework with full physics, sprites movements, and collision. The engine uses a scrolling camera of size 256×256 pixels (16×16 tiles), centered on Mario and clamped to the level bounds. Enemies are only spawned and begin moving once the camera reaches their tile position; until then they remain frozen at their spawn locations. Sprites that move more than 64 pixels beyond the camera horizontally, or fall below the level, are despawned.

Before simulation, three static pre-checks must pass: the empty tile ratio must exceed 0.5, tube tiles must be properly paired per row, and the ratio of enemies not standing on solid ground must be below 0.1. If any pre-check fails, the simulation is skipped entirely and all gameplay metrics default to zero.

When the pre-checks pass, a Mario AI agent plays the level to produce gameplay metrics. We evaluate under two solver configurations: *auto*, where a heuristic agent (always moves right and jumps) attempts the level first and falls back to A* search if it fails to complete; and *astar*, where the A* agent is used directly. The A* agent uses heuristic $h = (x + 10 \cdot v_x - x_{\text{root}}) - 10^6 \cdot \text{damage}$, where x is Mario's current pixel position, v_x is his horizontal velocity, and x_{root} is his position at the start of the search. The velocity term biases the search toward states with rightward momentum, while the large damage penalty causes the agent to actively avoid enemies rather than engage them. The simulation produces four key gameplay metrics: completion

percentage, enemies killed (excluding enemies that fall off ledges on their own), coins collected, and voluntary ground jumps (stomp bounces do not count).

- Quality: The quality score combines structural validity and gameplay completion:

$$q = \frac{q_{\text{tube}} + q_{\text{empty}} + q_{\text{noise}} + q_{\text{fenemies}} + 4 \cdot \text{complete}}{8} \tag{1}$$

  where q_{tube} rewards zero tube layout issues, q_{empty} rewards an empty tile ratio above 0.5, q_{noise} penalizes horizontal tile transition noise relative to a reference profile (only computed when $q_{\text{tube}} = 1$ and $q_{\text{empty}} = 1$), q_{fenemies} rewards zero floating enemies, and complete $\in [0, 1]$ is the fraction of the level traversed by the gameplay simulation AI agent. Each component uses a trapezoidal reward function in $[0, 1]$. Completion is weighted $4\times$ more than any structural component, emphasizing playability.
- Controllability: The controllability score measures whether the gameplay agent's simulation outcomes match sampled target values for three metrics: enemies killed, jumps, and coins collected:

$$c = \frac{c_{\text{enemies}} + c_{\text{jumps}} + c_{\text{coins}}}{3} \tag{2}$$

  Each component uses a trapezoidal reward centered on the target value with an error margin of $\lfloor 0.1 \cdot v_{\max} \rfloor$, where $v_{\max}$ is the maximum of the control range (4 for enemies, 6 for jumps, 3 for coins at width 32). But since all error margins evaluate to 0 for the map size of width 32, exact matches are required for a perfect score, making controllability particularly challenging. Only trials where the gameplay agent completes the level (complete ≥ 1.0) contribute to the controllability score.
- Diversity: The diversity score compares the gameplay agent's trajectory heatmaps between two levels rather than the level grids themselves. Per-frame pixel positions of Mario are converted to tile-grid visitation counts on a 16×32 heatmap matching the level dimensions. The L_1 distance between the two heatmaps, normalized by total frame count, is then passed through a trapezoidal reward with plateau at 0.4:

$$d = f_{\text{reward}}\left(\frac{\sum |V_1 - V_2|}{\sum V_1 + \sum V_2},\ 0,\ 0.4,\ 1.0 \right) \tag{3}$$

  where V_1, V_2 are the visitation heatmaps. This measures whether different levels produce meaningfully different play experiences, not just different tile arrangements.

Prompt Details

Although prompt content is problem-specific, all tasks use the same core template: agent role and goals, current level representation, evaluation metrics, score breakdown, termination status, available tools, strategy instructions, and a structured JSON response format. Depending on the configuration, the prompt may also include a tool-call budget and optional extra instructions. For Super Mario Bros., we provide more information in the prompt because the environment is dynamic. Specifically, the prompt further includes an explanation of gameplay-based evaluation and a simulation trajectory map summarizing agent behavior. In multi-turn settings, prompts are additionally prefixed with feedback from the previous step, including accept/reject status, score changes, metric deltas, and tool execution summaries.

Example Prompt From Binary Problem (Step 2)

````
You are an AI agent optimizing a binary maze level. Your goal is to achieve a longest shortest path of ~256 while
maintaining exactly 1 connected region.

## Current Level (rows are y, columns are x, 0-indexed):
```
....................................############################
##.............#########...########.############################
##..................................############################
###............#################################################
###.......######################################################
########..######################################################
########..####.#################################################
.............#.#################################################
.#######.....#.#.###############################################
.########.####.#.###############################################
.########......#.###############################################
.#############.#....############################################
.#############.#.##.############################################
.###...........#..#.############################################
.################.#.############################################
........##...........................###########################
.######.............##############.#############################
.######.............##############.#############################
````
````

```
....................##############.#############################
#######.............##############.#############################
#######.............##############.#############################
#######.............##############.#############################
#######............................#############################
#######..........###############################################
#######.##.###...###############################################
#######.#..###...###############################################
#######....###...###############################################
#######....###.#################################################
....###....###.#################################################
....###....###.###.#############################################
....###....###.###.#############################################
...................#############################################
```
Legend: '.' = empty (passable), '#' = wall (blocked)

Current Level Metrics:
  - path: 92 (goal: ~256)
  - num_connected_regions: 1 (goal: ~1)
  - traversable: yes

Score: -164.00 (path=92, regions=1, w_path=1.0, w_regions=10.0)

Termination Status:
Changes: 410/4096 (10.0% used, 3686 remaining)

Available Tools:

calculate_stats
Description: Calculates statistics for the current level including path (the longest shortest path between any two empty tiles) and num_connected_regions (number of separate connected regions of empty tiles). Returns metrics, targets, and gaps to help guide optimization.
Parameters:
  - level_string (string) (optional): Optional: ASCII level representation to evaluate. If not provided, evaluates the current level.

place_tile
Description: Places a tile on the level. Supported tile types: empty, wall. Supports three modes:
- 'single': place one tile at (y, x).
- 'line': place a straight horizontal or vertical segment. Specify the endpoint using ONE of: (a) direction ('up'/'down'/'left'/'right') + length, (b) end_y + end_x for an explicit endpoint, (c) end_x only (end_y defaults to y, drawing a horizontal line), or (d) end_y only (end_x defaults to x, drawing a vertical line). Example horizontal line: {mode:'line', y:5, x:0, end_x:10} Example vertical line: {mode:'line', y:0, x:3, end_y:8}
- 'rect': fill a rectangle from (y, x) to (end_y, end_x). Requires end_y and end_x.
Parameters:
  - mode (string) (required): Mode of operation: 'single' for one tile, 'line' for a segment, 'rect' for rectangle
  - tile_type (string) (required): Type of tile to place: empty, wall
  - y (integer) (required): Row coordinate (0-indexed). For line/rect, this is start_y.
  - x (integer) (required): Column coordinate (0-indexed). For line/rect, this is start_x.
  - end_y (integer) (optional): Ending row coordinate. Required for rect mode. For line mode: if only end_y is given (no end_x), draws a vertical line from y to end_y at column x. If both end_y and end_x are given, draws a line from (y,x) to (end_y,end_x).
  - end_x (integer) (optional): Ending column coordinate. Required for rect mode. For line mode: if only end_x is given (no end_y), draws a horizontal line from x to end_x at row y. If both end_y and end_x are given, draws a line from (y,x) to (end_y,end_x).
  - direction (string) (optional): Direction to extend from (y,x) --- alternative to end_y/end_x for line mode. Must be paired with 'length'.
  - length (integer) (optional): Number of tiles to place in 'direction' (for line mode with direction). Must be paired with 'direction'.
  - filled (boolean) (optional): For rect mode: if true, fill entire rectangle; if false, only draw border

generate_bsp
Description: Ignores the current level and generates a new room-and-corridor layout using Binary Space Partitioning. Recursively splits the space into rectangles, carves a room inside each partition, and connects adjacent rooms with corridors. Works regardless of the current level state. Output has open rectangular rooms linked by narrow passages --- typically 1 connected region with moderate path lengths. Increase splits for more, smaller rooms (longer paths); decrease for fewer, larger rooms (shorter paths). Calling this again discards all previous progress.
Parameters:
  - splits (integer) (optional): Number of recursive splits (default 3)
  - min_width (integer) (optional): Minimum partition width (default 5)
  - min_height (integer) (optional): Minimum partition height (default 5)

generate_ca
Description: REFINEMENT tool: evolves the CURRENT level in-place using cellular automata rules --- does NOT discard previous work. Each iteration, every tile checks its 8 neighbors (Moore neighborhood): if wall-neighbor count <= solid_count the tile becomes wall; if empty-neighbor count >= empty_count the tile becomes empty. Effect: smooths noisy layouts into organic cave-like shapes by removing isolated tiles and filling small holes. IMPORTANT: has no effect on a blank (all-empty or all-wall) level --- needs a mix of wall and empty tiles to work. Best used after generate_random or generate_digger to smooth rough output. Fewer iterations = subtle smoothing; more iterations = stronger smoothing toward large blobs.
Parameters:

````
  - iterations (integer) (optional): Number of CA iterations (default 10)
  - solid_count (integer) (optional): Neighbor threshold for becoming/staying wall (default 2)
  - empty_count (integer) (optional): Neighbor threshold for becoming empty (default 6)

## generate_connect
Description: REFINEMENT tool: post-processes the CURRENT level in-place to fix connectivity --- does NOT discard
previous work. Finds all connected empty regions, removes regions smaller than smallest_region_size (fills them
with wall), then connects remaining regions with straight corridors. Effect: reduces num_connected_regions toward 1
and increases path length by linking previously isolated areas. IMPORTANT: has no effect on a blank (all-empty)
level since it is already one region. Best used as a final pass after any generator (especially generate_random,
generate_ca, or generate_digger) to ensure the level is fully connected. Can also be called after manual place_tile
edits that may have split the level.
Parameters:
  - smallest_region_size (integer) (optional): Minimum region size to keep; smaller regions become wall (default 5)

## generate_digger
Description: Ignores the current level and generates a new cave layout using a random-walk digger. Starts from an
all-solid grid at a random position and carves empty tiles by walking in random directions, occasionally carving
rooms. Stops when the fraction of empty tiles reaches stop_size. Works regardless of the current level state.
Output is a single naturally-connected cave with organic, irregular shape --- guaranteed 1 connected region. Higher
stop_size = more open space (shorter paths); lower stop_size = tighter tunnels (longer paths). Follow up with
generate_ca to smooth rough edges. Calling this again discards all previous progress.
Parameters:
  - change_prob (number) (optional): Probability of changing walk direction (default 0.15)
  - room_prob (number) (optional): Probability of carving a room instead of a single tile (default 0.01)
  - room_size (integer) (optional): Half-size of carved rooms (actual size is 2*room_size+1, default 3)
  - stop_size (number) (optional): Stop when empty tile fraction reaches this value (0.0-1.0, default 0.3)

## Instructions:
1. Analyze the current level and its metrics
2. Plan edits to improve the score (move metrics toward goals described above)
3. Execute tool calls to modify the level
4. You may call calculate_stats to evaluate changes before committing

## Response Format:
Respond with a JSON object of one of these types:

### STEP - To execute tools:
```json
{
  "type": "STEP",
  "rationale": "explanation of your reasoning",
  "plan": "high-level plan for improvement",
  "tool_calls": [
    {"tool_name": "place_wall_segment", "parameters": {...}},
    {"tool_name": "calculate_stats", "parameters": {}}
  ],
  "acceptance_hint": "hint about whether to accept changes"
}
```

### PROPOSE_SKILL - To propose a new tool:
```json
{
  "type": "PROPOSE_SKILL",
  "rationale": "why this tool would help",
  "skill_spec": {
    "name": "tool_name",
    "description": "what it does",
    "parameters": {},
    "implementation_hint": "how to implement"
  }
}
```

### STOP - To terminate optimization:
```json
{
  "type": "STOP",
  "rationale": "why stopping now",
  "final_notes": "observations about the result"
}
```

Respond with ONLY the JSON object, no additional text.
````

Example Prompt From SMB Problem (Step 61)

```
You are an AI agent optimizing a Super Mario Bros level. Your goals are:
1. Create a completable level (Mario can reach the end)
2. Have proper tube/pipe structures (tubes must span 2 tiles properly)
3. Achieve ~3 enemies killed, ~5 jumps performed, ~3 coins collected during gameplay simulation
```

4. Minimize horizontal noise for smoother gameplay
5. Add coins and question blocks for rewards

How Evaluation Works
A Mario AI agent (A* solver) plays your level from left to right. The metrics below reflect the **gameplay outcome**, not the tile layout:
- **enemies_killed**: Number of enemies the player stomped or hit with shells during play. Enemies that fall off cliffs on their own do NOT count. Simply placing enemy tiles does not guarantee kills --- enemies must be on the player's path where they cannot be avoided.
- **coins_collected**: Number of coins the player picked up during play. Coins must be on or near the traversal path.
- **jumps**: Number of voluntary jumps the player performed from the ground. Bouncing off enemies (stomp bounces) does NOT count. More gaps and elevated platforms force more jumps.
- **complete**: Whether the player reached the end flag (1.0 = yes).

To increase enemies_killed, place enemies on narrow ground platforms that the player must cross (not on optional platforms). To increase coins_collected, place coins along the main path or on mandatory platforms. To increase jumps, create gaps in the ground and elevated platforms that force the player to jump from the ground --- enemy stomps do not count as jumps.

Current Level (rows are y, columns are x, 0-indexed):
```
................................
................................
................................
................................
................................
................................
...................GGGG.G.......
...................BBBB.BB......
................................
.................C..............
..........GCGCGG...........GC...
..........BBB.BB..........BBB.?.
.........................X......
...C.TT..C...............X....C.
.CGCGTT..................X.GGGG.
XXX.XXX.X.................XXXX.X
```
Legend: '.' = empty, 'X' = solid floor, 'L' = ladder, 'B' = brick, '?' = question block, 'T' = tube, 'C' = coin, 'G' = goomba, 'K' = koopa, 'Y' = spiny

Simulation Trajectory Map (last evaluation)
```
................................
................................
....................***.........
...................**..*........
..................**....*.......
..................*.....*.......
.............*...!.GGGG.G**....*
.............**.**.BBBB.BB*...**
........**..*.*.*.........*..*..
.......**.*.*.*.*C.........*.*..
...**..*..GCGCGG...........GC...
..*..*.*..BBB.BB..........BBB.?.
.**..**..................X......
**.C.TT..C...............X....C.
.!GCGTT..................X.GGGG.
XXX.XXX.X.................XXXX.X
```
Legend: '*' = Mario's path (empty tiles only), '!' = enemy killed here
Note: Enemies move during simulation. '!' marks where Mario was when the kill happened, not the enemy's spawn position. Enemies that were avoided or not reached are still shown at their original tile positions (G/K/Y).

Current Level Metrics:
  - complete: 100.0% (goal: 100%)
  - enemies_killed: 2 (goal: ~3)
  - coins_collected: 3 (goal: ~3)
  - jumps: 4 (goal: ~5) [stomp bounces: 2, not counted]
  - tube_issues: 0 (goal: 0)
  - empty_ratio: 88.5%
  - noise: 0.363 (goal: minimize)
  - simulation: ran
  - playable: yes

Simulation Results
The Mario AI agent played through the level. Here is what happened:
- Completion: 100% (reached the flag)
- Enemies killed: 2 (goomba stomped at tile [1, 14], goomba stomped at tile [17, 6])
- Coins collected: 3
- Jumps performed: 4 (stomp bounces: 2, not counted as jumps)
- Mario traversed from column 0 to column 31

````
Enemy behavior during simulation:
- Goomba (G): Walks horizontally, reverses on walls. Speed ~1.75 px/frame.
- Koopa (K): Green koopa walks horizontally, falls off cliffs. Becomes a kickable shell when stomped.
- Spiny (Y): Walks horizontally like goomba but CANNOT be stomped (hurts Mario). Must be avoided or killed with
shell/fireball.

## Score: 70.00 (complete=100%, tubes=0, enemies_killed=2, coins_collected=3, jumps=4)

## Termination Status:
Changes: 89/1536 (5.8% used, 1447 remaining)

## Available Tools:

## place_tile
Description: Places a tile on the level. Supported tile types: empty, solid, ladder, brick, question, tube, coin,
goomba, koopa, spiny. Supports three modes:
- 'single': place one tile at (y, x).
- 'line': place a straight horizontal or vertical segment. Specify the endpoint using ONE of: (a) direction
('up'/'down'/'left'/'right') + length, (b) end_y + end_x for an explicit endpoint, (c) end_x only (end_y defaults
to y, drawing a horizontal line), or (d) end_y only (end_x defaults to x, drawing a vertical line). Example
horizontal line: {mode:'line', y:5, x:0, end_x:10} Example vertical line: {mode:'line', y:0, x:3, end_y:8}
- 'rect': fill a rectangle from (y, x) to (end_y, end_x). Requires end_y and end_x.
Parameters:
  - mode (string) (required): Mode of operation: 'single' for one tile, 'line' for a segment, 'rect' for rectangle
  - tile_type (string) (required): Type of tile to place: empty, solid, ladder, brick, question, tube, coin,
  goomba, koopa, spiny
  - y (integer) (required): Row coordinate (0-indexed). For line/rect, this is start_y.
  - x (integer) (required): Column coordinate (0-indexed). For line/rect, this is start_x.
  - end_y (integer) (optional): Ending row coordinate. Required for rect mode. For line mode: if only end_y is
  given (no end_x), draws a vertical line from y to end_y at column x. If both end_y and end_x are given, draws a
  line from (y,x) to (end_y,end_x).
  - end_x (integer) (optional): Ending column coordinate. Required for rect mode. For line mode: if only end_x is
  given (no end_y), draws a horizontal line from x to end_x at row y. If both end_y and end_x are given, draws a
  line from (y,x) to (end_y,end_x).
  - direction (string) (optional): Direction to extend from (y,x) --- alternative to end_y/end_x for line mode.
  Must be paired with 'length'.
  - length (integer) (optional): Number of tiles to place in 'direction' (for line mode with direction). Must be
  paired with 'direction'.
  - filled (boolean) (optional): For rect mode: if true, fill entire rectangle; if false, only draw border

## Instructions:
1. Ensure the level is completable (Mario can traverse from start to end)
2. Place solid ground (X) for Mario to walk on
3. Add platforms using bricks (B) and question blocks (?)
4. Place enemies (G, K, Y) on solid ground along the player's path --- enemies the player can avoid will not count
5. Place coins (C) along the main traversal path --- coins the player never reaches will not count
6. Avoid creating impossible jumps or dead ends

## Response Format:
Respond with a JSON object of one of these types:

### STEP - To execute tools:
```json
{
  "type": "STEP",
  "rationale": "explanation of your reasoning",
  "plan": "high-level plan for improvement",
  "tool_calls": [
    {"tool_name": "place_tile", "parameters": {"mode": "single", "tile_type": "solid", "y": 15, "x": 1}}
  ],
  "acceptance_hint": "hint about whether to accept changes"
}
```

### PROPOSE_SKILL - To propose a new tool:
```json
{
  "type": "PROPOSE_SKILL",
  "rationale": "why this tool would help",
  "skill_spec": {
    "name": "tool_name",
    "description": "what it does",
    "parameters": {},
    "implementation_hint": "how to implement"
  }
}
```

### STOP - To terminate optimization:
```json
{
  "type": "STOP",
  "rationale": "why stopping now",
````
````

````
  "final_notes": "observations about the result"
}
```

Respond with ONLY the JSON object, no additional text.

## Extra Instruction:
Make level with two elevated platforms, requires travelling to upper platform to solve
````

Example Feedback Prefix to LLM Agent From SMB Problem (Step 61)

```
[Previous step result: REJECTED - rejected (score: 70.00 -> 50.00)]
Metrics change from your last edit:
  enemies_killed: 2 → 3 (target: ~3)
  coins_collected: 3 → 1 (target: ~3)
  jumps: 4 → 3 (target: ~5)
  complete: 100.0% → 100.0% (target: ~1.0)
  tube_issues: 0 → 0
  noise: 0.363 → 0.178
Tool results: 16 calls, 16 succeeded, 102 tiles changed
```
```