Mechanic Miner 2023: Reflection-Driven Game Mechanic Discovery Powered by Go-Explore

Niels NTG Poldervaart
Graduation Thesis

Media Technology MSc program
Leiden University
May 2024

Thesis advisors: Rob Saunders, Michael Cook

***Abstract*—We introduce Mechanic Miner 2023, a co-creative game design tool that suggests game design ideas by exploring the source code of the game itself instead of relying on predefined domain knowledge. Exploration is driven by an genetic algorithm which generates, evaluates and evolves simple two-state game mechanics that use code reflection to manipulate a property of the game's source code during gameplay. Evaluation is performed by an automated game-playing agent driven by Go-Explore, a state-of-the-art algorithm for automated game-playing. We demonstrate how Mechanic Miner 2023 can generate a diverse set of novel solutions for a simple 2D puzzle platformer within a time frame that suits iterative creative work.**

***Index Terms*—automated game design, computation creativity, mixed-initiative co-creative tools, procedural content generation, game design, game development, genetic algorithm, code reflection, artificial intelligence, game AI**

I. Introduction

In one form or another, Artificial Intelligence (AI) has been part of games for decades. This includes Procedural Content Generation (PCG), which is used both as part of the player experience as well the development of games itself. In both cases PCG is used for creating various types of game content such as level architecture and visuals. Procedural content generation systems that output game rules and mechanics aren't a common part of commercial game development, despite these receiving plenty of attention in academics [1][2][3][4][5][6][7][8][9][10][11]. Game mechanics are essential to the design of any game, with novel combinations of systems and rules being particularly appealing to game developers, especially those in independent game development. Many systems for generating rules and mechanics depend on predefined domain-specific data to ensure the validity and sensibility of the game design concepts they produce. However, constructing a model that encodes domain knowledge can be labor-intensive. And since games itself are usually not built from domain knowledge alone, human designers must translate the system's output into the game, making these systems less suited for real-life iterative game development[12].

Mechanic Miner (MM13) by Michael Cook, Simon Colton, Azalea Raad and Jeremy Gow. takes a different route by using the source code of the game and game engine itself as domain to search for game design ideas. It explores this space using an genetic algorithm, where it creates game mechanics using code reflection—the ability for the code to examine and modify itself during runtime—which are tested and evaluated by a game-playing algorithm and subsequently evolved[1].

In this paper we demonstrate the capabilities of a system called *Mechanic Miner 2023* (MM23), which re-implements parts of MM13. It differs in that MM23 is completely focused on the discovery of game mechanics alone, not level design as well. Instead of breadth-first search, the game-playing agent is driven by a the state-of-the-art Go-Explore algorithm[13]. And it is implemented in the popular Unity game development environment and is made publicly available as open source software under the MIT license on GitHub[1]. Unity's modular structure opens up a very large design space as well as faster game mechanic discovery. By analyzing its output from running it on a template game we will attempt to demonstrate the potential of the Mechanic Miner 2023 concept as a mixed-initiative co-creative tool[14].

In the following section we will elaborate on the background of this research, which includes the relationship between AI and games from a historical perspective, a brief taxonomy of Procedural Content Generation with examples of its application and a primer on Automated Game Design. The related work section discusses Mixed-Initiative Co-Creative tools, the common methods by which Automated Game Design systems encode game design and the merits of using source code as a design space. It will also describe MM13 in-depth where it applies to MM23. In the methodology section we explain the implementation of MM23 itself and setup of the experiment. The results of the experiment are shown and discussed in the results & discussion section, including case studies on some of the subjectively novel results and a discussion on MM23 viability as the basis of a fully fledged co-creative tool. Finally we conclude and suggest directions for future research.

[1]Source code repository for Mechanic Miner 2023 available on GitHub: https://github.com/Niels-NTG/Mechanic-Miner-2023

II. Background

A. AI & Games

In Western scientific circles, the game of Chess was for a long time seen as a test of one's intellect. Mastery of the game requires logical and strategic thinking. Building a computer program that could play it on a high level could be deemed intelligent, or rather an "artificial intelligence". Chess was therefore quite influential among early generations of computer scientists.

While the number of possible sequences of board states in classic games such as Checkers, Chess and Go are enormous, these are still theoretically finite and therefore computable. The challenge lies in finding the right algorithm to navigate through this astronomically large web of possible states. The creation of such an "intelligent" program became an important frontier in computer science and became one of the driving forces of artificial intelligence research for many decades[15]. This led to many discoveries with relevance far beyond the application to games. Notable examples include the *Minimax* tree search algorithm, initially discovered by John von Neumann in 1928[16], and later utilized by computer science pioneer Alan Turing in 1950 to automate the playing of Chess[17]. And the automated Checkers player by Arthur Samuel in 1957 using a method that is now known as *Reinforcement Learning*[18]. Both minimax and reinforcement learning are a foundational part of many AI systems today, for games as well as for completely different applications such as recommendation systems and robotics. Ever since there has been a continuous co-evolution of AI and games.

The goal of applying AI to games isn't only about creating systems that play to win. Nor do classic two player zero-sum abstract board games such as Checkers, Chess and Go encompass all types of games. Far from it. The medium is extremely broad and diverse. Some games are designed to be played by a single player, while others can be played by large groups. Players may compete with each other or work together and may not play to win but instead play to get a certain experience. Likewise, AI systems can be applied to many of these different aspects of the medium.

B. Procedural Content Generation (PCG)

Systems that incorporate Procedural Content Generation (PCG) methods can create content through algorithmic means. This enables automation of the creation of content that otherwise has to be created fully by hand. There is a large diversity of PCG methods, each differing in its degree of autonomy, controllability, determinism and adaptability.

PCG systems see a lot of use in games. Both as part of the player experience and as part of the game development process (the procedural content workflow). The types of content generated for games can be categorized in six distinct domains: level architecture and terrain, visuals, audio, narrative and rules and mechanics[4][19].

PCG can be used to generate a practically infinite variations of game content, which can greatly enhance a game's replayability value. This use case can be seen as early as *Beneath Apple Manor* (Don Worth, 1978) and *Rogue* (Michael Toy and Glenn Wichman, 1980) and has been increasingly common in games ever since. Around the year 2010 this application of PCG exploded in popularity among independent game developers. Exemplary games are *Spelunky* (Derek Yu, 2008) and *The Binding of Isaac* (Edmund McMillen and Florian Himsl, 2011), both taking queues from *Rogue*. With the rise of Large Language Models (LLM) it has become feasible to synthesize narrative content as well. For instance, *AI Dungeon* (Nick Walton, 2019) is a text adventure game which uses an LLM to generate characters, environments and scenarios for the player on-demand instead of relying on pre-authored content. Prototypes such as *Inworld Origins* (Inworld AI, 2023) and *NEO NPC* (Ubisoft, 2024) take it a step further by LLM-powered systems that embodied by 3D characters who can perceive and interact with their virtual environment.

PCG can also be used to save on storage space. Instead of having to store and load pre-made game assets, content can be generated on-demand by PCG methods. A deterministic PCG system can generate a whole game world from a single seed value, which allowed *Elite* (David Braben and Ian Bell, 1984) to fit a galaxy-sized world onto a single floppy disk and the voxel-based worlds of *Minecraft* (Mojang Studios, 2011) go on forever.

Game developers use various PCG tools to automate the creation of content that could otherwise be a very labor intensive task. These tools can be highly specialized such as the creation of vegetation with *SpeedTree* (IDV, 2002), character models with *MetaHuman* (Epic Games, 2021) and landscapes with *Terragen* (Matt Fairclough, 1999) or *World Machine* (World Machine Software, 2008). In large game productions it is the job of the technical artist to tie all these systems together using tools such as *Houdini* (Side Effects Software, 1996).

1) PCG for Game Rules and Game Mechanics: In the context of using PCG to generate game rules and game mechanics, we define game rules as what frames the playing experience. This could be the set of conditions for the game's win and fail state, or to forbid certain actions under certain conditions. These cannot be changed during gameplay. Game mechanics, also known as procedures, are the methods and actions available to the player to achieve the objectives of the game, such as being able to jump in *Super Mario Bros.* (Nintendo, 1985)[20]. Rule sets and associated mechanics tend to follow certain patterns that fit within the conventions of the game's genre. Games in the 2D-platformer genre for instance are expected to have a jump action. Just as with other types of game content, the rules and mechanics commonly associated with a genre may change over time.

Games which allow the player to change the rules of the game itself during gameplay, such as the puzzle game *Baba is You* (Hempuli Oy, 2019), *SuperMash* (Digital Continue, 2020) and *Mosa Lina* (Stuffed Wombat, 2022), do not in a strict sense actually change the rules and thus the framing of the game itself. Instead this ability is a game mechanic that changes the working of other game mechanics.

Most known PCG methods for generating rules and mechanics for the development of games are found in academic research. Of note is the *Ludi* system[2], which used an evolutionary algorithm to evolve rules and mechanics represented in a grammar by evaluating the depth and complexity of the game. The system "invented"—or rather discovered—the boardgame *Yavalath*[2][3], which became a moderately commercial success.

C. Automated Game Design (AGD)

Automated Game Design (AGD) is the area of study and the engineering practice of creating PCG systems that take an active role in the making of games by creating, editing and critiquing multiple types of game content simultaneously[21].

1) Brief History of AGD: Historically, AGD systems were primarily focused on generating rules and mechanics that fit within the domain of an game or a specific type of game. An early example of such a system is Pell's 1992 *Metagame*[22], which generates rules for symmetric Chess-like games.

Many of these early systems were primarily focused on generating rules and mechanics for games where this type of content comprise the entirety of the game's design; meaning it outputs abstract zero-sum boardgames that are reminiscent of Chess, Checkers, Go, etc. In many ways this mirrors the history of game AI as a whole, where most of the attention in the field was concentrated on creating AI systems for playing these same types of abstract zero-sum boardgames with the goal of outsmarting human opponents.

Since then AGD systems such as *Game-O-Matic*[7][8], *ANGELINA*[9][10], *Germinate*[5][6] and *Gamika*[23][24] incorporated PCG subsystems for types of game content such as visuals and narrative in addition to rules and mechanics, making these AGD systems able to handle multiple creative tasks at once.

III. Related Work

This section reviews research in the areas of mixed-initiative co-creative tools in the context of both game development in general and AGD specifically. Here we take a closer look at how AGD systems encode their output and discuss the merits of two different types of approaches: game description languages and using the source code domain. Finally we discuss *Mechanic Miner* (MM13)[1] in-depth where it relates to Mechanic Miner 2023.

A. Mixed-Initiative Co-Creative Tools

In any type of creative task, including game development, the tools to perform the task vary in terms of the amount of initiative the human designer needs to take in order to use it. For example, game engine editing tools such as *Unity* (Over the Edge Entertainment, 2005), *Unreal Editor* (Epic MegaGames, 1996) or *Hammer* (Valve Software, 2004) automate certain aspects of game development, yet still require a lot of proactive initiative from the user to do most of anything. At the same time the user has a lot of control over the output of the task.

In contrast to this there are tools where the computer takes most of the initiative. These tools are often specialized for a very specific tasks, such as the creation of the 3D models of vegetation (*SpeedTree*) or large-scale landscapes (*World Machine*). They do not require proactive input from the user after submitting initial parameters, nor does the user have any direct control over the output.

When the tool requires initiative from both the human designer and the computer in equal parts, this can be classified as a Mixed-Initiative Co-Creation (MI-CC) tool. This "dialog" between human and machine has great potential to foster creativity[14].

Examples of MI-CC systems applied to game development are *Tanagra*[25] and *Sentient Sketchbook*[26], which are level architecture PCG systems that are directly reactive to suggestions from the user.

Historically, the initiative in AGD tools were with the computer, not the user [7][8][9][10]. This somewhat limits the utility of these tools as part of the game development process, since these are designed to output a final product; not necessarily something that can be iterated upon further as part of a continuing creative process[27]. The *Puck* project addresses this shortcoming by having the system communicate to the user what it is currently working on as well being able to respond to the user's input at any time[11]. To facilitate the dialog between system and user, both need to understand a common language. In the case of *Puck* and many other AGD systems this is achieved by using a Game Description Language.

B. Game Description Languages

To encode domain knowledge, AGD systems are commonly built around a Game Description Language (GDL): a high-level description of a game's design that is readable by both humans and the AGD system via an interpreter. Works that incorporate GDLs include the aforementioned *Metagame*, with its own pseudo programming language[22], and *Ludi* with uses a grammar to encode its output[2]. On special note is the *Video Game Description Language* (VGDL)[28][29], which aims to be a more universal GDL. It has been used in a variety of research, such as to generate levels[30], game mechanics[31], whole arcade games[32] and to aid in the development of general game-playing agents [33][34][35][36][37].

While text files written in a GDL syntax to describe a game's design has the potential to be highly portable, the level of abstraction is too high for it translate seamlessly to a conventional game development environment. For example, a game developer cannot use the GDL in an iterative way if there is a misalignment between the capabilities of the GDL and the game development environment. Nor can it easily integrate with a pre-existing project[12][38].

C. Code Domain Space

Often AGD systems are designed to explore the design space within the domain of a single existing game or a specific

[2]*Yavalath*, as listed on *BoardGameGeek*, the online board game database: https://boardgamegeek.com/boardgame/33767/yavalath

type of game. One major benefit of this approach, is that the output of such a system has a higher likelihood of being something sensible that fits within the expectations of the domain. On the other hand, this approach limits the potential to find novel and surprising ideas that lay outside of the restrictions of the current domain.

To address this issue we can instead use the code-base of the game and game engine itself as the domain of the AGD system. While this is more open-ended than using using domain knowledge of existing games, the system needs to be made aware of code specifications that would otherwise be only implicit to a human programmer. This can be achieved by simply restricting the search space or by using programming design patterns such as the *Entity Component System*[12].

D. Mechanic Miner

In 2013, [1] introduced *Mechanic Miner* (MM13), a system that explores a game's source code using an evolutionary algorithm to discover simple game mechanics. Through reflection—the ability of the program to modify itself at runtime—game mechanics can be generated programmatically without the system having knowledge of game design.[1].

1) Toggleable Game Mechanics: The building blocks of a game mechanic generated by MM13 are *Toggleable Game Mechanics* (TGM). For the creation of a TGM, a property in the game's code, including that of the underlying game engine, is chosen at random. Then a modifier that is appropriate for the data type of the selected property (`double`, `half` or `invert`) is chosen at random.

The TGM is an action the player can toggle at any time whilst playing the game. When the TGM is turned on, it applies the modifier to the selected property's current value. When the TGM is off, the inverse modifier is applied to that value. This doesn't mean that the value is perfectly reversible, since other parts of the environment can also affect the value of the property.

An example of a TGM could be a property with a numeric value denoting the gravity force applied to the player object with an `invert` modifier. When the TGM is toggled on the value goes from 1.0 to -1.0, inverting the gravity and enabling the player to walk on the ceiling of the level until the TGM is toggled off, setting the gravity force back to 1.0 which makes the player fall back towards the ground.

2) Genetic Algorithm & Evaluation: To search the design space MM13 uses a genetic algorithm[39] which runs for 15 generations, maintaining a fixed population size of 100 members, each representing a TGM. 10% of each generation are brand new TGMs, while the remainder of the population members gets subjected to either crossover if the TGMs share the same data type or mutation, whereby either the TGM's property gets changed to another field the same class of the game's code or the modifier gets randomly changed.

The fitness value of a TGM is determined through simulating gameplay inside a sample game level. The game playing agent does a breadth-first search through the possibility space, each time looking for actions that would change the player's position within the level. The final fitness value is equal to how much of the level's space the player agent was able to explore while still being able to complete the level.

3) Game Environment: The game to which the MM13 system is applied is a simple 2D platforming game with procedurally generated levels. The level generator outputs levels of a fixed size of 20 horizontal and 15 vertical tiles. A level is enclosed on all sides and contains a player starting position, a level exit and a variety of obstacles, some in the form of spikes that result in instant death of the player. The player has the ability to move left, right, jump and toggle the TGM at any time during gameplay.

The game is implemented in Flixel[3], a Java-based game engine that can be compiled for many different platforms.

4) Significance, Contributions and Limitations: The MM13 system shows that the code domain can be used as a design space to search for game design ideas. Even though the system has very limited domain knowledge, it is still able to come up with unexpected and novel game design ideas.

However, the implementation of the game-playing agent used to evaluate the game design ideas was a limiting factor. To combat a combinatorial explosion, breadth-first search algorithm was restricted in how much of the action space it could explore.

IV. Methodology

This paper introduces *Mechanic Miner 2023* (MM23): a system that re-implements parts of MM13[1] with a number of enhancements. Firstly, MM23 does not implement the procedural level generator which co-evolves levels with discovered game mechanics. While Automated Game Design (AGD) systems can often create multiple types of game content simultaneously, doing both would make it difficult to evaluate the game mechanic generation capabilities of the system. Secondly, it's implemented in Unity, a popular and robust game development environment. Thirdly, the number of supported data types and modifiers for TGMs has been expanded compared to MM13. This greatly increases the possibility space the system can explore. Finally, the game-playing subsystem is driven by the state-of-the-art Go-Explore method[13].

A. Game Environment

MM23 is implemented using Unity[4]. This is a very popular game engine and development environment used by AAA game studios as well as small independent game developers, hobbyists and researchers. The editor is available for MacOS, Windows and Linux and projects made with it can be compiled to a wide variation of platforms.

Projects in Unity are compartmentalized into one or more *scenes*. Objects contained within a scene are *game objects*. Properties of a game object are defined by its components. Aside from the always required `Transform` component, which defines the object's translation, rotation and scale, game

[3]https://flixel.org
[4]https://unity.com

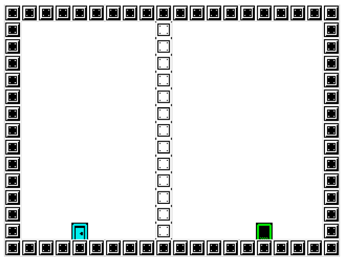
Fig. 1: **Level A: “Wall”**: impassable wall between the level start and exit.

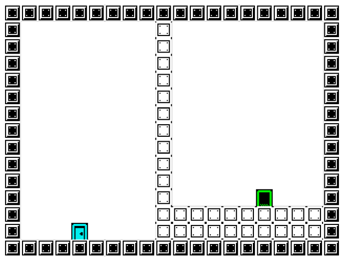
Fig. 2: **Level B: “Wall + Elevation”**: impassable wall between the level start and an elevated exit.

objects can have any number of components[5]. Unity comes with a wide range of component types, such as components that add rigid body physics or add a collider to the object. For creating any custom behavior that goes beyond what Unity has to offer out-of-the-box users can create their own components by writing C# code. All components can be accessed via its C# API as well as its own user interface visible in the editor. The UI is updated continuously to reflect the state of the scene.

The popularity of Unity as a game development tool and its robust object-oriented architecture makes it a good foundation for the development of MM23.

MM23 uses a simple 2D platformer as its template game. The player's goal is to navigate to the level exit by means of jumping, moving left and right and triggering a special action: the Toggleable Game Mechanic (TGM), see sectionIV-B. The levels are of a fixed size, are fully enclosed and always contain a starting and exit location. The level can contain solid obstacles that cannot be passed through under normal circumstances. It can also contain spikes which reset the player to the starting location when touched. The level architecture is aligned on a grid of squares. The player itself can move through the level in a continuous manner and is subject to simulated 2D rigid body physics.

The experiments are run with 6 different human-authored levels (see Figs 1–6). All of these levels are impossible to complete when only using “conventional” actions (move left, move right and jump), but may be possible if these are used in conjunction with one or more TGMs.

Due to the inherently stochastic nature of genetic algorithms, the system runs 40 times for each level. Runs are per-

[5]Unity user manual on components: https://docs.unity3d.com/2023.1/Documentation/Manual/class-GameObject.html

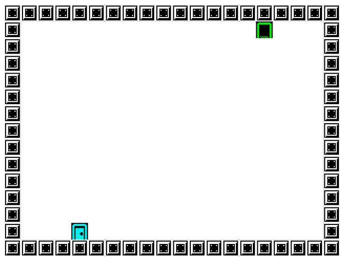
Fig. 3: **Level C: “Ceiling”**: level exit is located on the ceiling.

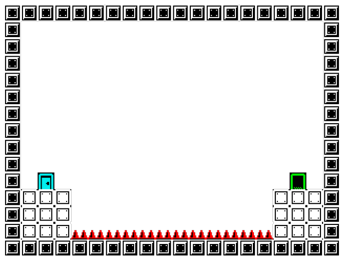
Fig. 4: **Level D: “Deadly River”**: level start and exit are on either banks of a wide “river” made of deadly spikes.

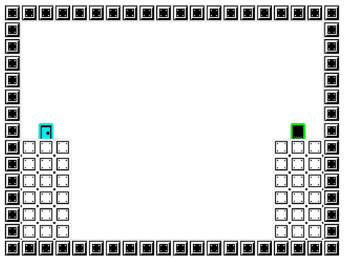
Fig. 5: **Level E: “Ravine”**: level start and exit are on either sides of a ravine.

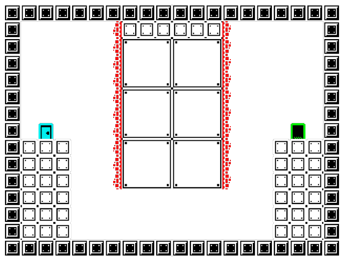
Fig. 6: **Level F: “Ravine + Spikes”**: level start and exit are on either sides of a ravine and there is an obstacle with spiked walls hanging from the ceiling.

formed with Unity version 2023.1 on a desktop PC equipped with an AMD Ryzen 5 3600, a consumer mid-range 6-core CPU[6], and 16 gigabytes of RAM.

B. Toggleable Game Mechanics

The TGMs are implemented the same way in MM23 as described in MM13 with minor changes.

To better guide the system towards discovering functional TGMs through the complex code domain space of the Unity game engine, the list of properties TGMs can be generated from has been limited to a user-defined list of components. For this experiment this includes the following:

- Transform, collider and rigid body component on the Player object.
- Grid layout and transformation of the level as a whole.
- Transform, collider, rigid body components for each layer that make up a a single level:
  - Outer walls of the level.
  - Obstacles and platforms inside of the level boundaries.
  - Level exit.
  - Spikes (if present).

Properties of the Unity game environment that aren't part of a component belonging to a game object in a currently loaded scene cannot be accessed programmatically while the game is running and therefore cannot be used as part of a TGM.

To extend the scope of the design space two more modifier types have been added in addition to the three from MM13 (`double`, `half` and `invert`) making for the following five modifiers:

- `double`: doubles the value, mirrors `half`.
- `half`: halves the value, mirrors `double`.
- `invert`: invert the value, mirrors with itself.
- `add`: adds one unit to the value, mirrors `subtract`.
- `subtract`: subtracts one unit from the value, mirrors `add`.

Additionally, the data types which can be used for a TGM has been extended beyond the primitive data types supported by MM13 (i.e. numeric and boolean types) to include more complex data types that are commonplace in Unity: vectors, quaternions, matrices, rectangles and enums.

When generating a new TGM, the generator picks a random field from a random component. Then it tests if it has a compatible data type, if it isn't read-only and if applying any of the modifiers to it causes the value to change.

C. Genetic Algorithm

Like MM13, MM23 makes use of a genetic algorithm to explore the design space. Genetic algorithms are a type of evolutionary algorithm in which individuals in a population are represented by a genotype of a set of genes. These get "evolved" over a number of generations by probabilistically selecting individuals after evaluating their fitness, and then combining individuals using methods such as gene crossover[39].

[6]AMD product description https://www.amd.com/en/product/8456

The genetic algorithm in MM23 is implemented using the C# package Genetic Sharp[40]. It evolves a fixed population of 100 members for 15 generations. It may terminate earlier if the development of the fitness value stagnates. The evolution of the genotypes, their fitness scores and evaluation simulation data are recorded for further analysis.

1) Genotype: The genotype of a population member consists of 4 genes, represented as text strings:

1) name of `GameObject`
2) name of `Component`
3) key of property in `Component`
4) modifier type (`double`, `half`, `invert`, `add`, `subtract`)

2) Selection: When selection is performed, the individuals are sorted on their fitness score. The top 10% highest scoring are selected (elite selection), while the 10% lowest scoring members are marked to be replaced with completely new genotypes. Roulette wheel selection is used on the remaining population members.

3) Evaluation: The fitness of a population member is based on the fraction of grid tiles within the bounds of the level the player is able to visit before finding the level exit. If the player agent isn't able to find the level exit before a fixed iteration limit is reached, the fitness is automatically zero. We find this to be a reasonable proxy to determine how much of a challenge the TGM provides the player while still being able to complete the level. A mechanic that enables the player to finish the level in the blink of an eye (e.g. instantly teleporting to the level exit) does not pose an interesting challenge for a human player, while a mechanic that encourages exploration does.

To perform the evaluation, a simulation is ran in an instance of the level that is isolated from other members of the population. To control the player agent, MM23 implements the exploration part of the Go-Explore algorithm[13], described in the next section.

D. Go-Explore

Games can have a high degree of complexity, requiring the player to think many steps ahead. One way to combat the immense branching factor is to use Reinforcement Learning (RL), whereby an agent learns a behavior (a *policy*) from receiving reward or penalty (negative reward) signals during training. There have been several high-profile cases in which RL-based game-playing systems were able to play on a level near or beyond that of highly-skilled human players[41][42][43].

One of the difficulties of engineering an RL-based system is the reward function. It can take a lot of trial-and-error to get right. If the reward signal is too dense—for example, the Euclidean distance between the agent and the game's ultimate goal—the agent could learn a very greedy policy, where it cannot find an alternative route if there are obstacles on the path to the goal. Likewise, if the reward is too sparse—for example, only reward the agent if it reaches the game's

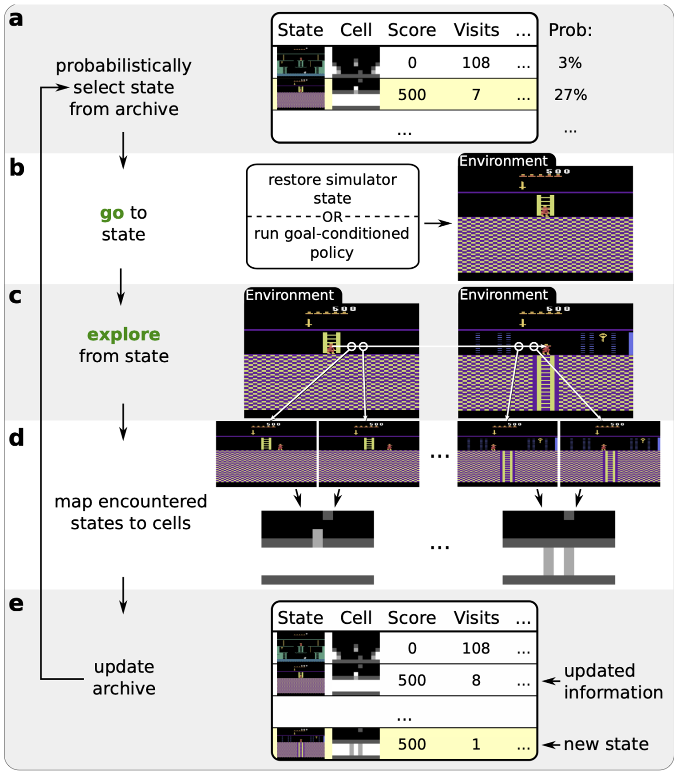

Fig. 7: Overview of the Go-Explore system. This figure has been copied from its original paper[13] with written permission of its first author.

ultimate goal—the agent may learn a policy that largely relies on random actions.

Go-Explore[13] attempts to address this problem by taking some ideas from classical planning algorithms, except that it does not attempt an exhaustive search of the space. The main idea being that while the agent takes actions in the environment it builds an archive of the resulting states. The archive is a list of cells, where each cell represents a set of similar states. A cell stores only a single state, which can get replaced if a state with a shorter trajectory (number of actions leading to that state) is found for that cell. After completing an exploration roll-out (explore), it probabilistically selects a cell from the archive, restores the agent to that state (go) and continues the exploration from there. See Fig. 7 for an overview of the Go-Explore algorithm.

The trajectories generated from exploring the environment with the Go-Explore method can be used as input to train a policy using methods such as imitation learning to drive the behavior of a game-playing agent. This is what the authors of Go Explore method call the *robustification phase*. In the case of MM23 however, only the exploration phase is implemented to control the game-playing agent during the simulation used to evaluate a genotype.

V. Results & Discussion

A. *Genetic algorithm and game-playing system performance*

1) Non-zero fitness population size development: The generational development of median population size of members with a fitness value above zero follows a distinct pattern for most levels (Fig. 8). At first the number climbs gradually, only to suddenly shoot up, after which is plateaus and gradually decreases. This pattern cannot be observed in levels D (Fig. 4) and F (Fig. 6). It's however possible that if the genetic algorithm would have kept running for longer than 15 generations the shape of the plots for these levels would have ended up being very similar.

2) Analysis of fitness evaluation method: In systems that make use of an evolutionary algorithm it is usually desirable that the fitness value has an upwards trajectory. Ideally the maximum possible value is reached before a set time limit or number of generations, allowing such a system to stop early. Fitness values in conjunction with the selection mechanism are a very substantial on how such an algorithm traverses the feature space. However, when looking at the non-zero fitness values coming out of MM23 it can be seen that the development of median fitness levels over the different runs has a very flat trajectory (Fig. 9).

Ideally the same genotype in the same environment should yield a fitness value that is nearly exactly the same every time. This however not the case in MM23 due to the non-deterministic nature of the game-playing algorithm part of computing the fitness value. Depending on the TGM + level combination there can be a lot of variance in the fitness value (Fig. 10). This creates instances where TGMs that were evaluated to have a high fitness value in one generation to suddenly disappear from the population just because in a later evaluation of the same genotype in the same environment the game-playing agent happens to perform poorly.

This makes it difficult to assess if the fitness metric (see IV-C3) chosen for MM23 is a useful estimate on how interesting a mechanic might be. For this reason the fitness values of the TGMs discussed in the case studies sectionV-B will be ignored.

This noisy signal could explain why the trajectory of the fitness levels is so flat (Fig. 9), as well as why in levels A (Fig. 1) and B (Fig. 2) the number of solutions with a fitness higher than zero declines gradually after peaking (Fig. 8). MM23's selection method selects the top 10% fittest individuals (*elite selection*) and replaces the bottom 10% with new individuals, leaving the rest to be selected using *roulette wheel* selection. The chance for any individual to be selected here depends on their fitness. But if there is such a high variance in fitness values between otherwise identical individuals this part of the selection system becomes a too unpredictable.

3) Unique gene count development: Looking at the median number of unique TGMs that had a fitness above zero (Fig. 11) shows that the system tends to converge on a single "best" solution instead of finding as many different feasible solutions

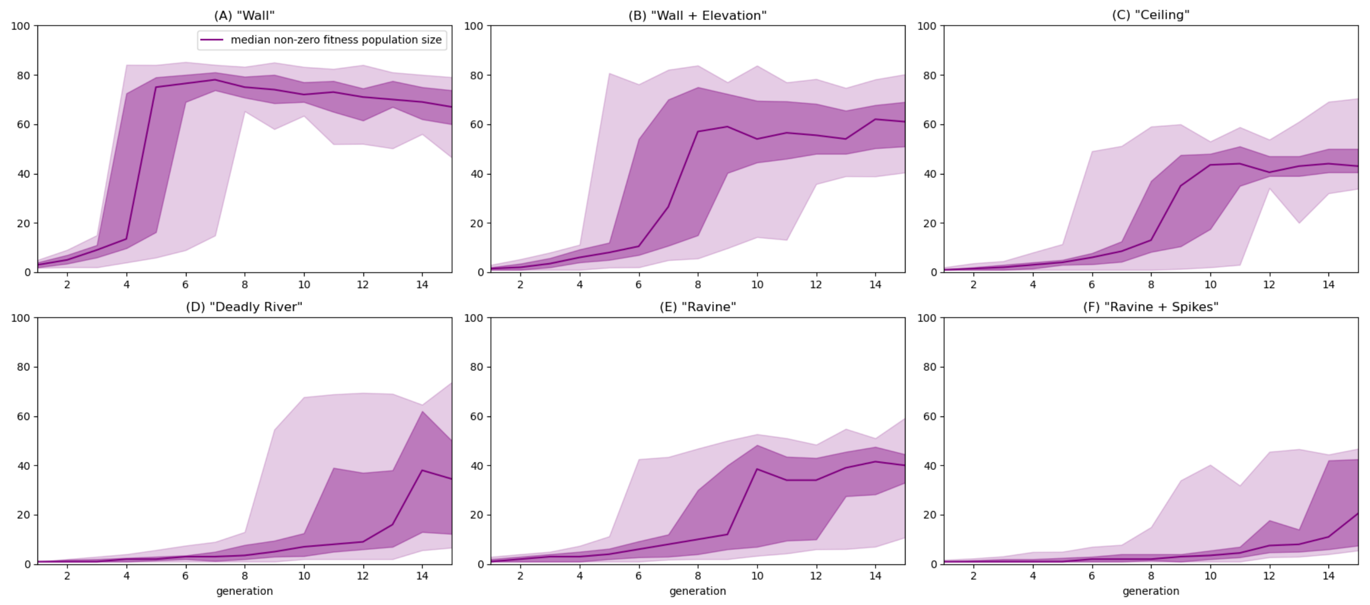

Fig. 8: Evolutionary development of the median and $5\% - 25\% - 75\% - 95\%$ percentile number of population members with a fitness above zero.

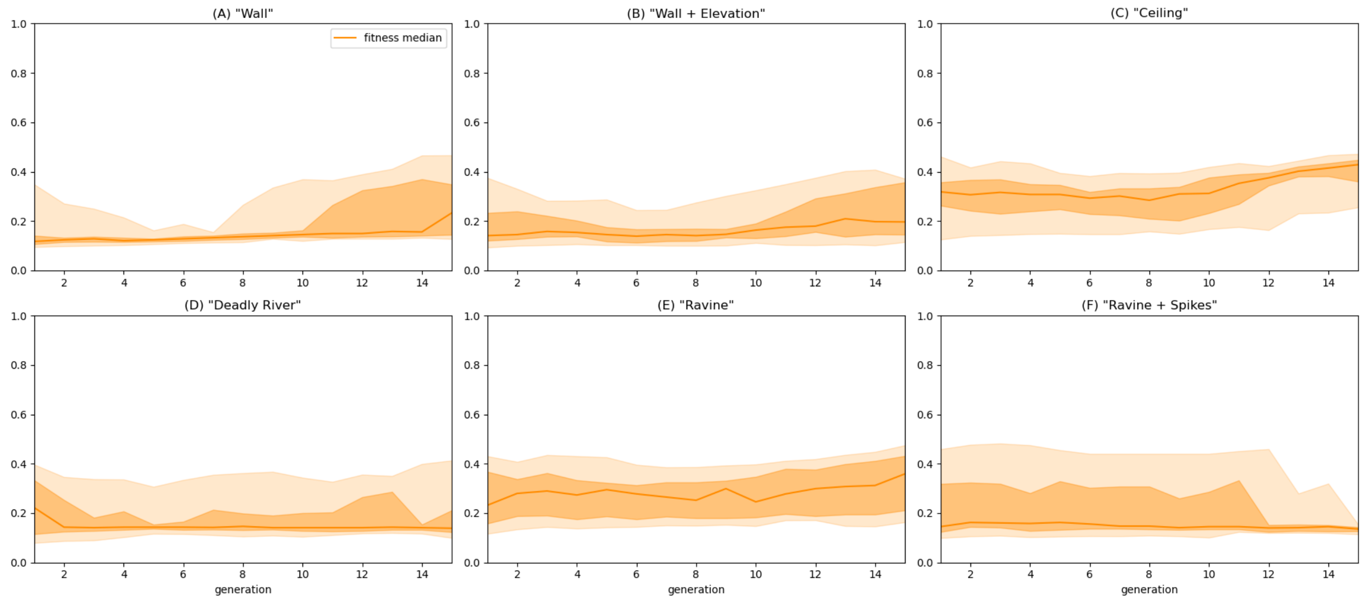

Fig. 9: Evolutionary development of the median and $5\% - 25\% - 75\% - 95\%$ percentile fitness values.

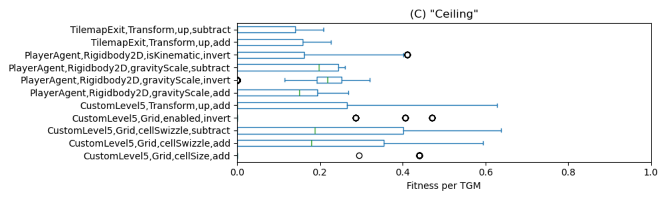

Fig. 10: Box-and-whisker plot of fitness values for all TGMs discovered in level C (Fig. 3) across 40 runs, excluding TGMs that never reach a fitness value larger than zero.

as possible. This is to be expected from a classic evolutionary algorithm like what was implemented in MM23, but in part may also be caused by the aforementioned inconsistent fitness values. An additional termination condition that stops the genetic algorithm when the number of unique solutions declines could be a way to prevent this from happening and help to ensure sufficient diversity among the population.

B. *Case studies*

Since the confines of the design space are known (see IV-B), the resulting TGMs can be grouped based on the combination

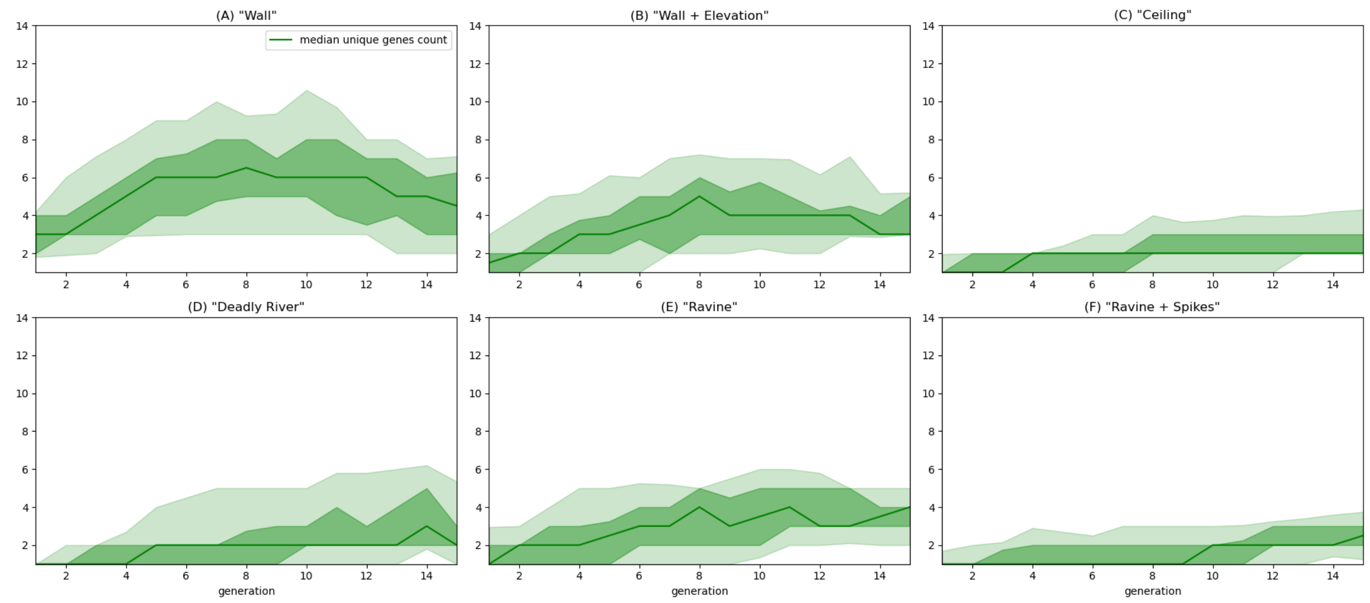

Fig. 11: Evolutionary development of the median number and $5\% - 25\% - 75\% - 95\%$ percentile of unique genotypes with a fitness value larger than zero across 40 runs for each level.

of the type of game object (player or level) and the type of component (transform, collider, etc.). When looking only at the TGMs which helped the game-playing agent to finish the level (a non-zero fitness), the results show quite a diverse group of solutions (Fig. 12). In the rest of this section various notable TGMs found in the data are discussed in detail.

1) Physics manipulation: TGMs that manipulate the effect of physics on game objects[7] were discovered in all levels. Especially in levels C (Fig. 3), D (Fig. 4) and E (Fig. 5) this was quite a common type of solution.

In level C for instance, a common solution to reach the exit located on the ceiling of the level is to invert the gravity scale acting on the player (`PlayerAgent`, `Rigidbody2D`, `gravityScale`, `invert`), making it fall towards the ceiling instead of the floor. This is very much reminiscent of the "gravity inversion" TGM discussed in the results of MM13[1].

In level E two common solutions were to subtract the mass of the player (`PlayerAgent`, `Rigidbody2D`, `mass`, `subtract`) or the gravity force acting on the player (`PlayerAgent`, `Rigidbody2D`, `gravityScale`, `subtract`) by 1, enabling the player to jump over much larger distances than would otherwise be possible.

Another common occurrence in level E was a TGM that changes rigid body of the outer walls of the level from static to dynamic (`TilemapOuterWall`, `Rigidbody2D`, `isKinematic`, `invert`). When activated the top of the level fall onto the banks of the ravine, creating a bridge for the player. While this seems like a novel and creative solution, playing the level manually as a human reveals it's not possible to clip through the level boundary with this TGM alone. It only works in the simulation due to an oversight in the implementation of the game-playing agent: when the agent restores to an earlier state, it only restores the state of the player object, not any other objects in the level.

2) Grid layout manipulation: TGMs that manipulate the grid layout[8] of the level itself were discovered in all levels. Very often TGMs of this type ended up being among the population in the final fifteenth generation of a run (Fig. 12).

In levels A (Fig. 1) and B (Fig. 2) a common solution is half the size of the tiles on the grid (`CustomLevel3`, `Grid`, `cellSize`, `double`), enabling the player to "slip through" the gaps between the tiles of the wall which is normally blocking access to the level exit (Fig. 13).

A TGM that halves instead of doubles the cell size was also a viable solution. This effectively halves the scale of the level without moving the player, making it so the player is now at the other side of the wall (Fig. 14). Another common grid manipulation TGM is those that change the way the grid tiles are arranged. Variants of this solution could be seen in every level, but it was especially prevalent in level C (Fig. 3), where it was used to "flatten" the level into a single row of tiles (Fig. 15).

3) Colliders: While collider components in Unity have a wide variety of properties[9], with a few exceptions the TGMs discovered related to colliders were all alike. All disabled a collider of an object in the level such that it would no longer be an obstacle for the player. A clear demonstration

[7]The component in Unity that controls the effects of physics on an object is called the `Rigidbody` https://docs.unity3d.com/2023.1/Documentation/Manual/class-Rigidbody.html

[8]Unity's `Grid` component reference: https://docs.unity3d.com/2023.1/Documentation/Manual/class-Grid.html

[9]Unity documentation on the `Collider` base class https://docs.unity3d.com/2023.1/Documentation/ScriptReference/Collider.html

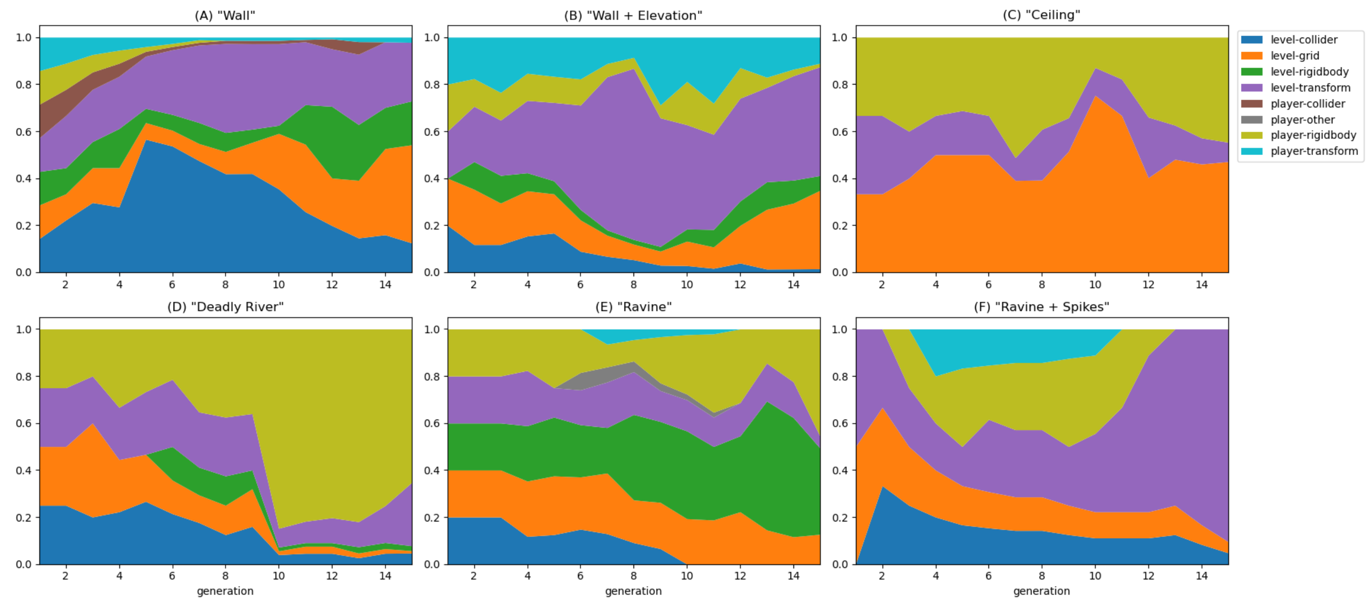

Fig. 12: Evolutionary development of the median distribution of TGMs types with a non-zero fitness value in 40 runs of the system for each level.

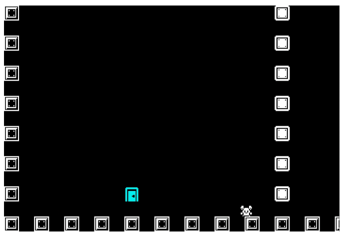

Fig. 13: Doubling the grid cell size with `CustomLevel3`, `Grid`, `cellSize`, `double` repositions all tiles on the grid while the objects within the tiles stay the same size.

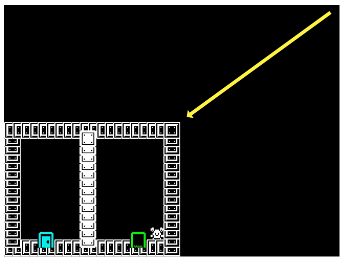

Fig. 14: Halving the grid cell size with `CustomLevel3`, `Grid`, `cellSize`, `half` repositions all elements on the grid to align with this grid of smaller cells while the objects in the cells stay the same size.

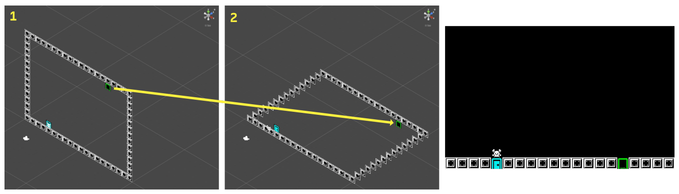

Fig. 15: With the TGM `CustomLevel5`, `Grid`, `cellSwizzle`, `add` the coordinate system of the grid is reordered such that the x-axis becomes the z-axis and visa versa. In essence this rotates the entire level 90 degrees along the global x-axis, flatting the entire level into a single row of tiles from the perspective of the game's camera. The level exit, previously on the ceiling (1), now can be easily reached now it's on the same horizontal line as the level start (2).

of this can be seen in level A (Fig. 1), where a TGM such as `TilemapPlatforms`, `TilemapCollider2D`, `enabled`, or `invert` disables collision on the wall separating the player and the exit when activated.

A more complex utilization of this mechanic can be observed in levels B (Fig. 2) and F (Fig. 6). Here the player can move inside of the obstacle underneath the level exit when the collider is disabled. If timed correctly, the player can get to the top of the obstacle by jumping and then enabling the collider again, making the obstacle solid once again. Jumping is not a required action however, since enabling the collider of the obstacle while the player is inside exploits a behavior of the game engine to resolve the invalid state of intersecting rigid bodies by pushing these apart. This depenetration algorithm applies a force between the center of gravity of the two bodies. So if the player happens to be above the center of mass of the obstacle the player will be pushed upwards towards the level exit.

In Unity colliders can be translated and resized relative to the object they are attached to. Only a small number of TGMs that make use of this property were discovered by MM23. A clear demonstration can be seen in level A (Fig. 1), where it

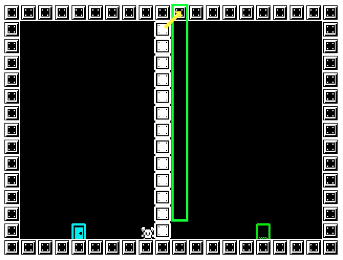

Fig. 16: `TilemapPlatforms`, `CompositeCollider2D`, `offset`, `add` adds 1 unit to the translation offset of the wall's collider (marked in green) creating a gap large enough for the player to move through.

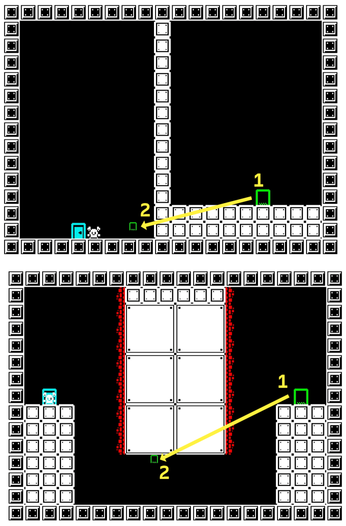

Fig. 17: `TilemapExit`, `Transform`, `localScale`, `half` divides the scale vector by 2 when activated. Since the origin of this object is at the lower-left corner of the level, the level exit (1) does get smaller and moves to the other side of level's main obstacle (2).

offsets the collider by one unit from the level's origin, creating a gap underneath the wall allowing the player to pass through (Fig. 16).

4) Transformation tricks: Solutions that translate, rotate or scale the player object or a part of the level occurred in every level. Especially in level F (Fig. 6), where solutions of this type were particularly dominant.

A common solution for the levels B (Fig. 2) and F (Fig. 6) was to half the scale of the level exit such that it moves to the same side of the wall as where the player starts (Fig. 17). Similar is one where the current position of the player gets doubled, effectively "teleporting" it to the other side of the wall and on top of the platform (Fig. 18), reminiscent of similar behavior seen in MM13.

In level A (Fig. 1) a TGM was found that could move the main obstacle of the level out of the way by essentially switching places with the player19.

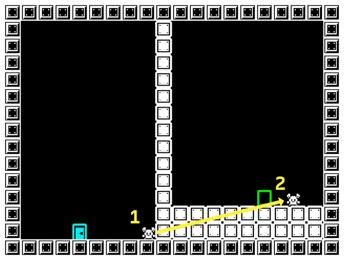

Fig. 18: `PlayerAgent`, `Transform`, `position`, `double` multiplies the player's current position vector by 2. If the player is positioned far enough from the lower-left corner of this level (origin of the player's coordinate system) (1), the x-component's value is high enough to move it to the other side of the wall when doubled (2).

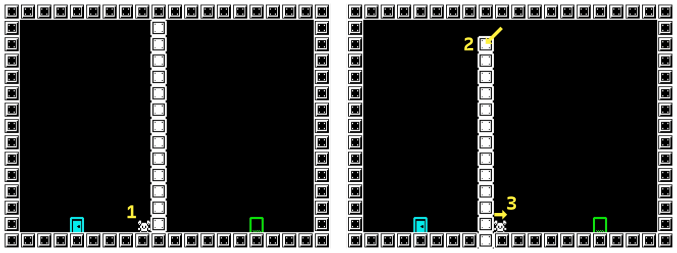

Fig. 19: When the player is standing next to the wall blocking the the path to the level exit (1), activating the TGM `TilemapPlatforms`, `Transform`, `localPosition`, `subtract` shifts that wall one unit towards the origin in the lower left corner (2). Since the colliders on both the player and the wall cannot overlap, the game engine resolves this by pushing the player to other side of the wall (3).

C. Mechanic Miner as a co-creative tool

A single run of the MM23 system, whereby a population of 100 genotypes is evaluated for at most 15 generations on a single level, takes approximately 8 to 15 minutes on a desktop PC with an AMD Ryzen 5 3600 (a consumer mid-segment 6-core CPU) and 16 gigabytes of memory. While it does utilizes all CPU cores, it cannot use these to their maximum capacity consistently. Unity requiring calls to the active scene to be executed on the main thread is a likely cause of this, resulting in other threads having to "wait their turn", limiting the amount of computations that can be done in parallel. While it's very likely there is plenty of room for optimizations that could further speed up the system, the current results already demonstrate that MM23 can be run on locally on an ordinary personal computer to output results in a reasonable amount of time. The ability to deliver solutions quickly an an important part of any co-creative tool.

Some of the solutions generated by MM23 seem to fit within what would be expected of a solution thought up by a human game designer given the same game and level architecture. For instance, altering how gravity acts on the player object to get across a wide deadly hazard as seen in level D (Fig. 4) is an example of one of those "obvious" solutions. MM23 also generated types of solutions that are far outside the conventions of the puzzle-platformer genre. Altering the arrangement or layout of the level's grid for instance was quite a common solution found for levels A (Fig. 1), B (Fig. 2), C (Fig. 3) and E (Fig. 5). From a human player's perspective this type of mechanic may however seem not very intuitive. It may require additional work on the part of the game designer to implement this game mechanic concept in a way that is more understandable to a human player.

Since Mechanic Miner is meant to be a co-creative tool, this isn't a problem. It isn't expected nor even desirable for a co-creative tool to output complete end-to-end solutions. Instead it should help the human designer in their creative process by providing them with novel and diverse ideas.

For Mechanic Miner to be useful as a co-creative tool it is expected to give suggestions that spark creativity on the side of the user. While measuring creativity is difficult, optimizing for the output to be as diverse as possible has shown to be a very effective way of getting to a desired target[44]. In terms of promoting diversity, the MM23 system has demonstrated to be able to output a diverse set of solutions for any problem it is given (Fig. 12). Genetic algorithms like the one used in MM23 however tend to be optimized to find a single "best" solution, not a diverse set of solutions. If there the termination condition of a 15 generation limit was removed, the number of unique solutions would most likely trend towards 1. This downwards trend could be seen with some levels, particularly with A and B (Fig. 11). This on to itself may be easily preventable by introducing an additional termination condition to the genetic algorithm that checks if the diversity the population is trending downwards. All though it might be more effective to change the selection criteria such that only unique individuals are taken into consideration, which could prevent a small set of genes from becoming dominant in the population.

Despite the limitation of the chosen search strategy, the results do demonstrate that utilizing the source code as a search space to generate game mechanics with code reflection is a feasible alternative to using game description languages or otherwise pre-constructed domain knowledge.

VI. Future Work

A. Game-playing agent

As was described in the results, the game-playing agent component of MM23 performs inconsistently between runs in exactly the same environment and starting conditions (see V-A2). For the purposes of an evolutionary algorithm where this game-playing agent is a key part of determining the fitness score, this noisy signal makes it difficult to evaluate the effectiveness of our fitness function, since the same genotype combined with the same environment yields different fitness score any time the simulation is ran.

There are a number of potential solutions for this. A record could be kept of the highest fitness values for each unique genotype. If the simulation yields a lower fitness value than previously recorded, use the recorded value. If not, update the record with the new value. Or alternatively, when a genotype is created that hasn't been seen before it is evaluated in multiple parallel running simulations to find the highest fitness value among the results from those. This approach has the added benefit of no longer having to re-run simulations for subsequent generations of the same genotype, which could drastically reduce the time it takes the system to run.

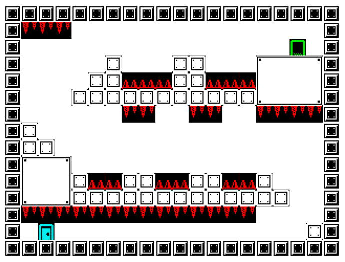
Fig. 20: **Level G: "Spiked S-bend"**: S-shaped upwards corridor with plenty of instant-death spikes along its ceilings and floors. Can be completed without any TGM, but doing so requires very precise actions.

When designing the experiment for this paper, one of the human-authored levels was ultimately excluded from the experiment due to our Go-Explore implementation not being able to complete this level, apart from a small number of "lucky" attempts (Fig. 20). This seems due to the sequence of specific actions required to complete it being too long for Go-Explore to find. Using Go-Explore to train a policy on an authored curriculum of levels prior to running the evolutionary algorithm could be a potential solution for these more complex game environments. It may help the game-playing agent to play through levels much more reliably, at the cost of the human game designer having to provide a curriculum of situations for the agent to train in, which in turn may hinder MM23's utility as a co-creative tool for quick iterative game development.

B. More complex game environments

To further explore the potential and limitations of the Mechanic Miner concept, we would like to apply it to more complex and open-ended games.

In Rogue and games like it for instance the end state is no longer a simple binary fail or pass, instead these games encourage the player to maximize their score. The score is calculated from multiple factors such as the progression through the different stages and by gathering various different resources. These things alone already allow for a wide variety of viable player strategies. A Mechanic Miner-based system could be used to synthesize new resources or abilities for the player to find. These could be evaluated using metrics that align with the intentions of the game designer.

C. Quality-Diversity search

Both MM23 and its predecessor MM13 use a fairly standard evolutionary algorithm as a means to explore the possibility space. Generally this type of algorithm starts with a diverse and random population to search for local optima, meaning that the total diversity in the population tends to trend downwards as it finds population members with higher and higher fitness values. Here the promotion of diversity is mainly a way to prevent the search from getting stuck on local optima in the search of the highest possible fitness value in the possibility space.

Quality-Diversity (QD) optimizations are different from classic evolutionary algorithms in that these aim to generate large collections of diverse yet high-performing population members. Novelty Search is relatively simple yet effective implementation of QD system, where it rewards population members for behavior that diverges from the rest of the population. While the classic fitness metric is still present, it it only used to determine stopping conditions, not selection[44].

Applying a QD method such as Novelty Search may be an effective way to counteract the downwards trending fitness and population diversity as was seen in some cases in our results (see V-A2). It would make Mechanic Miner a more effective co-creative tool if does more in-depth exploration of a diverse set of game design ideas before rejecting them in favor of something that seems more promising in the short term.

VII. Conclusions

In this paper we presented Mechanic Miner 2023 (MM23), a procedural content generation system that discovers game mechanics based on code reflection by exploring the source code of the game itself using an evolutionary algorithm, where the found game mechanics were tested and evaluated by an automated game-playing algorithm based on Go-Explore.

We've demonstrated that using the game's source code as a search domain for procedural content generation for game mechanics content is a viable alternative to using game design-specific data models such as game description languages. This makes MM23 more flexible and has the potential to integrate with a wide range of pre-existing game development projects.

We have shown MM23 can output diverse and novel game mechanics that fit with a pre-authored game design within a reasonable amount of time on an ordinary PC, making it suitable for use as a co-creative tool.

Acknowledgments

The first author of this paper would like to thank Tom Murphy VII for showing him the creative and whimsical side of computer science, every attendee of the 2021, 2022 and 2023 editions of the International Artificial Intelligence and Games Summer School, Jonathan Jørgensen, Pedro M. Fernandes, Christoph Salge, Peter van der Putten, the Generative Design in Minecraft (GDMC) community and the Leiden University Media Tech Master program.

References

[1] M. Cook, S. Colton, A. Raad, and J. Gow, "Mechanic miner: Reflection-driven game mechanic discovery and level design," in *Applications of Evolutionary Computation*, ser. EvoApplications'13. Springer, 2013, pp. 284–293.

[2] C. Browne and F. Maire, "Evolutionary game design," *IEEE Transactions on Computational Intelligence and AI in Games (CIG)*, vol. 2, pp. 1 – 16, 04 2010.

[3] C. Browne, *Yavalath*. Springer, 2011, pp. 75–85.

[4] A. Liapis, G. N. Yannakakis, and J. Togelius, "Computational game creativity," in *International Conference on Innovative Computing and Cloud Computing*, ser. ICCC'14, 2014.

[5] M. Kreminski, M. Dickinson, J. Osborn, A. Summerville, M. Mateas, and N. Wardrip-Fruin, "Germinate: A mixed-initiative casual creator for rhetorical games," in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, ser. AIIDE'20, vol. 16, no. 1, Oct. 2020, pp. 102–108.

[6] A. Summerville, C. Martens, B. Samuel, J. Osborn, N. Wardrip-Fruin, and M. Mateas, "Gemini: Bidirectional generation and analysis of games via asp," in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, ser. AIIDE'18, vol. 14, no. 1, Sep. 2018, pp. 123–129.

[7] M. Treanor, B. Blackford, M. Mateas, and I. Bogost, "Game-o-matic: Generating videogames that represent ideas," in *Workshop on Procedural Content Generation in Games*, ser. PCG'12. Association for Computing Machinery, 2012, p. 1–8.

[8] M. Treanor, B. Schweizer, I. Bogost, and M. Mateas, "The micro-rhetorics of game-o-matic," in *International Conference on the Foundations of Digital Games*, ser. FDG'12. Association for Computing Machinery, 2012, p. 18–25.

[9] M. Cook, S. Colton, and J. Gow, "The angelina videogame design system—part i," *IEEE Transactions on Computational Intelligence and AI in Games (CIG)*, vol. 9, no. 2, pp. 192–203, 2017.

[10] ——, "The angelina videogame design system—part ii," *IEEE Transactions on Computational Intelligence and AI in Games (CIG)*, vol. 9, no. 3, pp. 254–266, 2017.

[11] M. Cook, "Puck: A slow and personal automated game designer," in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, ser. AIIDE'22, vol. 18, no. 1, Oct. 2022, pp. 232–239.

[12] ——, "Software engineering for automated game design," in *IEEE Conference on Games*, ser. IEEE CoG'20. IEEE, 2020, pp. 487–494.

[13] A. Ecoffet, J. Huizinga, J. Lehman, K. O. Stanley, and J. Clune, "First return, then explore," *Nature*, vol. 590, no. 7847, p. 580–586, feb 2021.

[14] G. N. Yannakakis, A. Liapis, and C. Alexopoulos, "Mixed-initiative co-creativity," in *International Conference on the Foundations of Digital Games*, ser. FDG'14. Foundations of Digital Games, 2014.

[15] N. Ensmenger, "Is chess the drosophila of artificial intelligence? a social history of an algorithm," *Social Studies of Science*, vol. 42, no. 1, pp. 5–30, 2012.

[16] J. von Neumann, "Zur theorie der gesellschaftsspiele," *Mathematische Annalen*, vol. 100, pp. 295–320, 1928.

[17] A. M. Turing, "Digital computers applied to games," *Faster than thought: a Symposium on Digital Computing Machines*, 1953.

[18] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, 1959.

[19] G. N. Yannakakis and J. Togelius, *Artificial Intelligence and Games*. Springer, 2018, http://gameaibook.org.

[20] T. Fullerton, *Game Design Workshop : a Playcentric Approach to Creating Innovative Games, Third Edition*. CRC Press, 2014.

[21] M. Cook, "Getting started in automated game design," 2020. [Online]. Available: https://www.youtube.com/watch?v=dZv-vRrnHDA

[22] B. Pell, "Metagame: A new challenge for games and learning," *Heuristic Programming in Artificial Intelligence 3 - The Third Computer Olympiad*, 1992.

[23] E. J. Powley, S. Colton, S. Gaudl, R. Saunders, and M. J. Nelson, "Semi-automated level design via auto-playtesting for handheld casual game creation," in *IEEE Conference on Computational Intelligence and Games (CIG)*, ser. IEEE CIG'16, 2016, pp. 1–8.

[24] E. Powley, M. Nelson, S. Gaudl, S. Colton, B. Pérez Ferrer, R. Saunders, P. Ivey, and M. Cook, "Wevva: Democratising game design," in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, ser. AIIDE'21, vol. 13, no. 1, Jun. 2021, pp. 273–275.

[25] G. Smith, J. Whitehead, and M. Mateas, "Tanagra: an intelligent level design assistant for 2d platformers," in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, ser. AIIDE'10, 2010, p. 223–224.

[26] A. Liapis, G. N. Yannakakis, and J. Togelius, "Sentient sketchbook: Computer-aided game level authoring," in *International Conference on the Foundations of Digital Games*, ser. FDG'13, 2013.

[27] M. Cook and S. Colton, "Redesigning computationally creative systems for continuous creation," in *International Conference on Computational*

Creativity, ser. ICCC'18. Association for Computational Creativity (ACC), 2018, pp. 32–39.

[28] M. Ebner, J. Levine, S. M. Lucas, T. Schaul, T. Thompson, and J. Togelius, "Towards a video game description language," in *Artificial and Computational Intelligence in Games*, ser. Dagstuhl Follow-Ups. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013, vol. 6, pp. 85–100.

[29] T. Schaul, "A video game description language for model-based or interactive learning," in *IEEE Conference on Computational Inteligence in Games (CIG)*, ser. IEEE CIG'13, 2013, pp. 1–8.

[30] A. Khalifa, D. Perez-Liebana, S. M. Lucas, and J. Togelius, "General video game level generation," in *Genetic and Evolutionary Computation Conference*, ser. GECCO'16. Association for Computing Machinery, 2016, p. 253–259.

[31] A. Khalifa, M. C. Green, D. Perez-Liebana, and J. Togelius, "General video game rule generation," in *IEEE Conference on Computational Intelligence and Games (CIG)*, ser. IEEE CIG'17, 2017, pp. 170–177.

[32] T. S. Nielsen, G. A. B. Barros, J. Togelius, and M. J. Nelson, "Towards generating arcade game rules with vgdl," in *IEEE Conference on Computational Intelligence and Games (CIG)*, ser. IEEE CIG'15, 2015, pp. 185–192.

[33] J. Levine, C. B. Congdon, M. Ebner, G. Kendall, S. M. Lucas, R. Miikkulainen, T. Schaul, and T. Thompson, "General Video Game Playing," in *Artificial and Computational Intelligence in Games*, ser. Dagstuhl Follow-Ups, S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius, Eds. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013, vol. 6, pp. 77–83.

[34] D. Perez-Liebana, S. Samothrakis, J. Togelius, T. Schaul, S. M. Lucas, A. Couëtoux, J. Lee, C.-U. Lim, and T. Thompson, "The 2014 general video game playing competition," *IEEE Transactions on Computational Intelligence and AI in Games (CIG)*, vol. 8, no. 3, pp. 229–243, 2016.

[35] D. Perez-Liebana, S. Samothrakis, J. Togelius, T. Schaul, and S. Lucas, "General video game ai: Competition, challenges and opportunities," in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, ser. AIIDE'16, vol. 30, no. 1, Mar. 2016.

[36] R. D. Gaina, D. Pérez-Liébana, and S. M. Lucas, "General video game for 2 players: Framework and competition," in *Computer Science and Electronic Engineering (CEEC)*, ser. CEEC'16, 2016, pp. 186–191.

[37] M. Johansen, M. Pichlmair, and S. Risi, "Video game description language environment for unity machine learning agents," in *IEEE Conference on Games (CoG)*, ser. IEEE CoG'19, 2019, pp. 1–8.

[38] T. Duplantis, I. Karth, M. Kreminski, A. M. Smith, and M. Mateas, "A genre-specific game description language for game boy rpgs," in *IEEE Conference on Games (CoG)*, ser. IEEE CoG'21, 2021, pp. 1–8.

[39] J. H. Holland, "Genetic algorithms," *Scientific American*, vol. 267, no. 1, pp. 66–73, 1992.

[40] D. Giacomelli, "Geneticsharp," Jul. 2019. [Online]. Available: https://github.com/giacomelli/GeneticSharp

[41] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, and Others, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[42] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, and Others, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[43] P. Wurman, S. Barrett, K. Kawamoto, J. Macglashan, K. Subramanian, T. Walsh, R. Capobianco, A. Devlic, F. Eckert, F. Fuchs, L. Gilpin, P. Khandelwal, V. Kompella, H. Lin, P. Macalpine, D. Oller, T. Seno, C. Sherstan, M. Thomure, and H. Kitano, "Outracing champion gran turismo drivers with deep reinforcement learning," *Nature*, vol. 602, pp. 223–228, 02 2022.

[44] J. Lehman and K. O. Stanley, "Exploiting open-endedness to solve problems through the search for novelty," in *IEEE Symposium on Artificial Life*, 2008.