

Programmable Cellular Automata

Ahmed Khalifa^{1,3,*}, Muhammad Umair Nasir^{2,*}, Matthew Siper³, Steve James² and Julian Togelius^{3,4}

¹*Institute of Digital Games, University of Malta, 20 L. Esperanto, L-Imsida, Malta*

²*University of the Witwatersrand, 1 Jan Smuts Ave, Braamfontein, Johannesburg, 2017, South Africa*

³*Nof1, 665 Broadway, Brooklyn, NY 10012, USA*

⁴*Game Innovation Lab, New York University, 370 Jay St, Brooklyn, NY 11201, USA*

Abstract

Cellular automata is a local computation paradigm where complex behavior can arise from local interactions between simple functions. This paradigm has been used to explain many systems such as biological processes, traffic simulation, computer networks, etc. In games, cellular automata have been used in games such as SimCity and for the generation of spatial content such as caves or dungeons. However, creating effective local rules is hard and unintuitive. Cellular automata can be effectively evolved, but may still be hard to interpret. In this work, we introduce the concept of programmable cellular automata, where we represent the system as Python code. We also modularize the cellular automata into local functions and a decision function. Local functions take a local neighborhood and return a value, while the decision function takes the output of the local functions and decides the value of the next state. Separating the cellular automata into modules written in Python helps with understanding how these systems are working. We also explore adding global functions where they take the whole state and compute a function from it. We tested generating levels for three different games from the PCG Benchmark. The results showed that global functions decrease the number of iterations that cellular automata need to solve a problem, and that we cannot find solutions for some problems with purely local functions. Looking into the generated functions, we can see common functions that have been used in different experiments, which not only helps us understand the generator but also helps us understand these games better and what is important for them.

Keywords

Cellular Automata, Procedural Content Generation, Genetic Programming, Large Language Models

1. Introduction

The appeal of cellular automata is that simple local computation can create complex global phenomena. A single cellular automaton implements a very simple function, mapping the surroundings of a cell to the next state of the cell. But applying this function iteratively to every cell in a two- (or three-, or n-) dimensional lattice can yield surprisingly complex behavior. Most famously, the Game of Life [1] features an impressive menagerie of gliders, factories, blinkers, spaceships, and so on. The expressive powers of these 2^8 possible automata are indeed surprising. Perhaps even more surprising, it has been shown that the Game of Life is Turing complete, meaning that any algorithm can be implemented within a sufficiently large lattice of this automaton. The power of these simple automata has inspired some rather fringe theories, in particular Stephen Wolfram's idea that the universe itself is a cellular automaton [2]; this idea has inspired fiction such as Greg Egan's Permutation City [3].

More prosaically, cellular automata have proven useful tools for many tasks. CA have been used to model biological and chemical growth processes [4, 5], and to simulate, e.g., traffic [6] or computer networks [7]. Within games, cellular automata are one of the core abstractions within the SimCity game

Woodstock'22: Symposium on the irreproducible science, June 07–11, 2022, Woodstock, NY

*Corresponding author.

✉ ahmed@akhalifa.com (A. Khalifa); muhammad.nasir@wits.ac.za (M. U. Nasir); m@thenof1.com (M. Siper); steven.james@wits.ac.za (S. James); julian@togelius.com (J. Togelius)

🌐 <http://www.akhalifa.com> (A. Khalifa); <https://sdjames.me> (S. James); <http://julian.togelius.com> (J. Togelius)

🆔 0000-0002-7839-9432 (A. Khalifa); 0000-0002-2458-9599 (M. U. Nasir); 0009-0008-6570-8209 (M. Siper);

0000-0003-4366-4125 (S. James); 0000-0003-3128-4598 (J. Togelius)



© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

series [8]. They are also useful for level generation, where it has been shown that they can generate, for example, organic-looking cave systems [9].

The canonical CA is a purely discrete affair and can be visualized simply as a table, making the structure very easy to understand and edit for humans. However, many more expressive function spaces are possible. In particular, a relatively recent variant is the Neural Cellular Automaton (NCA) [10]. In an NCA, the state of a cell at the next time step is decided by a weighted sum of the neighboring cells passed through a nonlinear function. In other words, the cell acts as a neuron. An NCA can be seen as a single-layer convolutional neural network looped on itself.

While NCAs are significantly more expressive, at least per cell, than regular cellular automata, they are equally significantly harder to understand. Much like how a neural network is harder to understand than a lookup table.

Are there other ways of representing cellular automata that retain (or even increase) the expressivity from NCAs but are easier for humans to inspect and author? Yes. Program code. We can simply represent the functions in a modern high-level programming language such as Python. If the only input to the function is the state of the cells around the target cell and the output is the state of the cell at $t + 1$, this is still cellular automata, yet one that can be understood by anyone who can read the programming language.

In many cases, you want to learn or automatically construct your cellular automaton rather than building it by hand, in particular if you want to rapidly obtain a CA that fulfils a particular purpose, such as a level generator for a new game. If the new use case comes with an evaluation function (or verifiable reward, as it is also called), one can use an evolutionary algorithm to search for suitable automata. This works with all CA representations. However, with a programmatic representation, we can make use of the tremendous power of code-writing LLMs [11]. Agentic coding with prompts to improve the CA can work as a replacement for or complement to traditional mutation and recombination operators.

Another benefit of a programmatic representation is the potential for increased modularity. One could define a cellular automaton as not one but several functions that connect in a specified way, for example, with the output of one as input to another. That way, it is easy to understand, improve, and swap the functions separately. This increased modularity also allows us to include global functions. This is of interest because whereas there exist Turing-complete cellular automata (such as the Game of Life, see above), it seems to be very hard in practice to accomplish certain tasks with entirely local computation, particularly within limited time frames. For example, in many cases you may need a certain number (such as 1, 10, or more than 5) of cells with a particular activation; this requires some type of counter mechanism, which is fiendishly hard and inefficient to implement using only local computation.

In this paper, we describe an initial exploration of programmable cellular automata (PCA). We propose an architecture for PCAs which combines multiple local functions with, optionally, global functions that provide information about the current state and location that get fed to a decision function that decides the values for the next state.

2. Related work

Research in cellular automata and procedural content generation (PCG) focused on extending the algorithm capabilities to generate more complex content such as Minecraft buildings [12], dungeon levels [13, 14], spatial layouts [15, 16], terrain [17, 18], city layouts [19], gameplay mechanic [20], etc. This direction is mostly common in game development since the generated algorithms are constructive, so they are well suited for online generation [21].

Research in cellular automata has focused on evolving the rules such that they can compute some global value [22, 23]. This is due to the difficulty of designing local rulesets that can compute these global values. In PCG, this is about combining two paradigms: constructive approaches (cellular automata) [24] and search-based approaches [25]. In some of the work [26, 27], they evolved the cellular automata rules and starting state, and the number of iterations to generate levels or maps for video games. Evolving

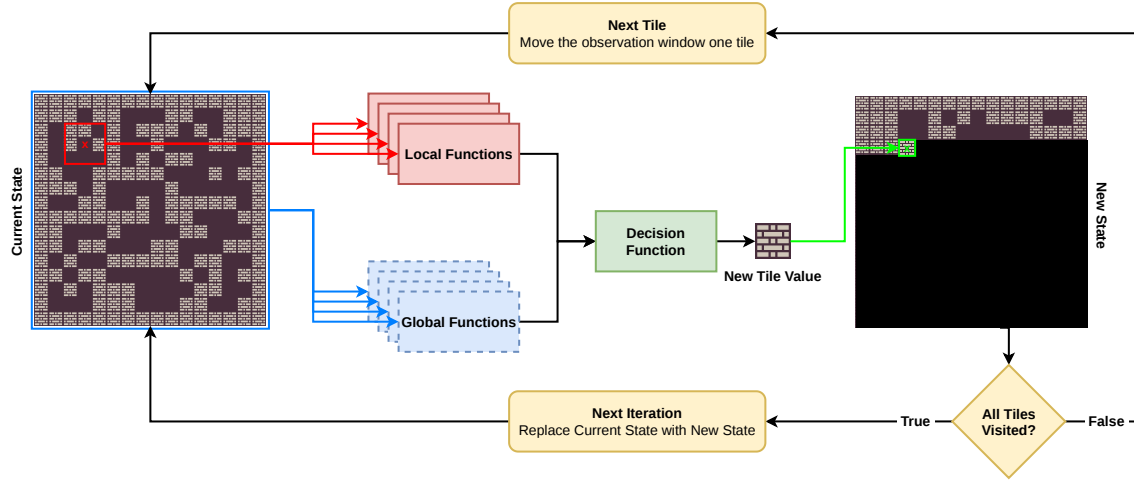


Figure 1: Programmable Cellular Automata follows the same loop as cellular automata, where the current state gets processed one tile at a time and produces a new state; then, in the end, the new state becomes the current state, and you repeat again. The difference can be seen in the middle part where we divided into two or three types programs: Local Functions (compute a function based on local observation), Global Functions (Optional: this is not traditional CA as it takes a full observation and compute a function), and finally Decision function (compute the final value for the current location in the new state using all the output from the other functions).

the rules, the starting state, and the number of iterations together means that the evolution is evolving an indirect representation of a single level instead of a generator. Others looked at cellular automata as a generator [28, 29, 30] and evolved the rules of the level generator for top-down games. Earle et al. [31] expanded the rule evolution by representing the rules as a neural cellular automata (NCA) and evolved it using a quality diversity algorithm and tested it on multiple games. Earle and Togelius [32] evolved NCAs that represent game mechanics for new environments to train RL agents. Sudhakaran et al. [33] evolved NCA to generate 3D voxel assets.

With the rise of LLMs [34], procedural content generation research focused on using LLMs as a generator to generate content [35] such as level generators [36, 37, 38, 39], game generators [40, 41, 42], story generators [43], etc. Another direction is that, instead of generating content or the solution, you generate code that can solve the problem. This is done with the help of a search/evolutionary algorithm. Romera et al. [44] used the LLM to evolve a scoring function to rank solutions such that the system can select the best candidate solution. Lehman et al. [45] evolved code that generates 2D robot designs in an open-ended evolution fashion [46]. Novikov et al. [47] used a quality-diversity algorithm with an LLM to evolve code that can solve multiple different problems. The LLM managed to write more efficient matrix multiplication code than the ones that have been used so far. Ma et al. [48] evolved a reward function for RL algorithms and tested it on 29 different environments, where in a lot of cases the trained RL agent with these evolved functions beats human-trained agents.

This paradigm of writing code is still new, but it works perfectly with the PCG domain, as the LLM agents can write a fast constructive generator that can be used directly in games. Siper et al. [49] used an LLM to write Python code as a constructive level generator [24] for 4 different games. They combined the LLM’s ability to write code with an evolutionary loop to find a generator that can generate consistent levels that pass the playability criteria. It also used the LLM to abstract and summarize the generated code into utility functions that other chromosomes can use to improve quickly. This work is inspired by their work, as we look at CA as modular pieces where each piece can be represented by a function that can be produced using any type of code generation, including LLMs.

3. Programmable Cellular Automata

Programmable Cellular Automata (PCA) is a new formulation of Cellular Automata (CA). Instead of defining the CA as a table of local transitions, it is represented as a program. In this respect, it is similar to Neural Cellular Automata (NCA) [10]. The difference is that, in NCA, all the local functions are represented as convolutional kernels, while the decision function is represented as a fully connected layer. This allows CA to be more expressive but, at the same time, not understandable in terms of its rules. PCA is a more general formulation of this where all these functions are represented as programs (such as program trees, Python code, neural networks, etc).

Figure 1 shows PCA, where at each step the system sends the local observation to all the local functions (instead of the convolutional kernels), and then the decision function (instead of the linear layer) takes all these local computations and decides on the cell value. The PCA iterates over all the cells and computes the new value to be part of the next state. After the PCA finishes iterating over all the cells, the next state replaces the current state of the whole lattice and a new iteration starts.

Although the CA paradigm is focuses on local computation, we are interested in extending it to include global functions and see their effect. The global functions take the whole state instead of a small local observation and try to compute a helping value. In this case, the decision function, at each step, takes the result of all the local computations (from the local functions) and, if available, the global state computations (from the global functions) and decides on the next cell value. This version of PCA is not fully local computationally, but has capabilities to do more complex operations within a small number of iterations.

There are a few constraints for PCA on all these programs:

- **Local Functions:** should take some form of local neighborhood (such as Von Neumann or Moore neighborhood) from the current state and produce a value(s).
- **Global Functions (Optional):** should take the whole current state and produce a value(s).
- **Decision Function:** should take all the different values from the local functions and global functions (if existing) and return a valid value(s) for the next cell state.

The most important thing is that the local/global functions shouldn't change anything in the state, and the decision function shouldn't take anything from the state itself; instead, it should take the computed values from these functions.

Finally, CA are usually *Synchronous*, which means that the changes in each cell only depend on the current state and only replace the current state with new values when everything is being computed. We retain this convention, but we note that global functions could benefit from being *Asynchronous*. *Asynchronous* means that any modification by the decision function to a cell will affect the next computation of the global function. For example, imagine you want a function that counts the number of players, and we are using a *Synchronous* global function; if there are no players, the value will be 0 for all the local interactions, which makes the decision function add a player for each local location. If it is *Asynchronous*, after execution at each local location, the system can update the global count so the next location will know that there is a player has already been added.

4. Automated Discovery of Programmable Cellular Automata

Instead of writing programmable cellular automata as humans, we can automate the process of writing them utilizing genetic programming [50, 51]. In this section, we go over the different parts that are needed to allow genetic programming/evolutionary algorithms to explore new PCAs.

4.1. Representation and Initialization

The chromosome is an array of l local functions, g global functions, and an decision function, where l is the number of local functions (at least 1) and g is the number of global functions, which can be 0 if you decide not to have any global functions. The functions can be represented as any type of program,

it can be represented as trees, strings that represent Python code, or machine learning programs (like neural networks). In this work, we focus on programs represented as Python code because it is easy to understand, and because it makes it easy to utilize current Large Language Models (LLMs) to generate better initial functions than random generation of trees or random neural networks, which can speed the search.

4.2. Variational Operators

For variational operators, programmable cellular automata allow for crossover and mutation operators. Because the chromosome is represented as an array of programs, we can apply crossover easily using uniform crossover, one-point crossover, or other types. In this project, we are using uniform crossover where each program can be selected from one of the parents with equal probability. Of course, swapping a local function or global function might not have the same effect as swapping the decision function (since the decision function affects the state changes). Exploring the effect of that is out of scope of this work.

For mutation, we only need a way to generate a new function; this can be random sampling of the code tree, or using an LLM to generate new code, or adding some noise to neural network weights, etc. Since, in the current work, we represent the functions as Python code, we are using an LLM to generate new functions. This can happen in two ways:

- **Random Sampling:** We prompt the LLM to generate the selected function (local, global, or decision function) given its restrictions (input parameters and/or output) and use the output directly from the LLM. In principle, we would not need to use an LLM at all for this, but any type of code generation as long as it can generate code that respects the given constraints.
- **Context Sampling:** Similar to the above, but besides the restrictions of the function, we also give it context about the rest of the functions (local, global, and decision functions) that it works with and the fitness score that it achieves using these functions. This is harder to achieve without an LLM. Using context helps to guide the LLM towards generating different functions that could improve the fitness of the overall chromosome.

4.3. Fitness

For evolving the PCA, we need to evaluate its output on multiple inputs. We ran the PCA n times on multiple different inputs. For each run, we ran the PCA for m iterations, and if during these iterations it found a solution, we stopped; if not, we evaluated all the states using our fitness function and returned the maximum value. So the fitness function for each PCA is defined by equation 1.

$$f = \begin{cases} 1 + \frac{1}{n} \sum_{i=0}^n D_i & \text{if } \forall_i \in \{0, 1, \dots, n\}, Q_i = 1 \\ \frac{1}{n} \sum_{i=0}^n Q_i & \text{otherwise} \end{cases} \quad (1)$$

$$Q_i = \max_{k \in m} (q_k) \quad (2)$$

$$D_i = \min_{k \in n} (dist(i, k)) \quad (3)$$

where f is the final fitness, D_i is the minimum distance of all other outputs from the different runs (as defined in equation 3), Q_i is the highest reached quality of each output from the runs over its iterations (as defined in equation 2), q_k is the quality of the state k in run i , and $dist(i, k)$ is a distance function between two outputs from the PCA.

The fitness function integrates not just the quality (Q_i) but also the diversity of the output content (D_i). It uses diversity as a cascaded fitness [52], which means that if the output is never fully functional in all n runs, diversity won't contribute to the fitness. Diversity in a lot of problems is very important since we want our PCA to respond differently to different inputs, but if that is not the case (where the more important thing is the solution being just functional), then you can remove that term from equation 1.

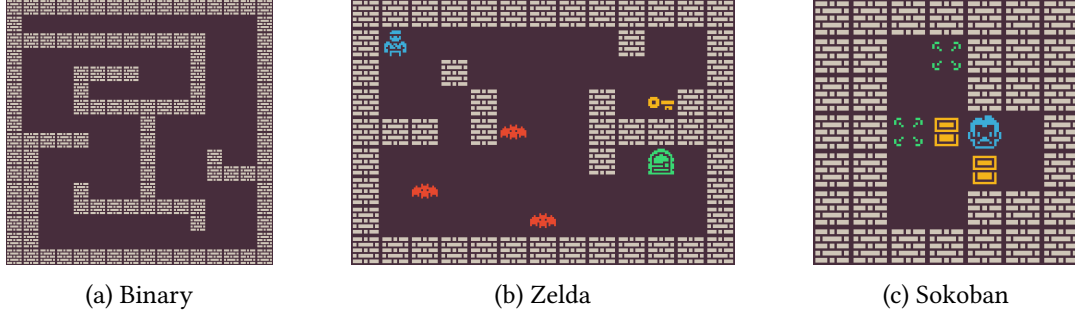


Figure 2: Example of human-designed levels for all the 3 game domains. Each of these levels satisfies the corresponding quality metric.

5. Experiments

In this paper, we test PCA on the domain of Games. Each PCA represents a level generator [9] for different problems from the PCG Benchmark [53]. For each of the problems, we use the quality and distance metrics provided by the benchmark. Figure 2 shows the selected problems from the PCG Benchmark, which are:

- **Binary** is a simple 14x14 (without borders) maze made of solid and empty tiles where the final maze needs to be fully connected and has a path of at least 28 tiles (as shown in figure 2a).
- **Zelda** is a simple 11x7 (without borders) dungeon-crawling game inspired by The Legend of Zelda (Nintendo, 1986) dungeon rooms. The goal of the player is to grab a key and reach a door without getting killed by enemies. The levels need to have at least 1 player, 1 key, 1 door, and 3 enemies. The solution length (getting the key then getting the door) is at least 18 moves (as shown in figure 2b).
- **Sokoban** is a simple 5x5 (without borders) puzzle game based on a Japanese puzzle game with the same name (Thinking Rabbit, 1981). The goal of the game is for the player to push all the crates onto the targets. The levels need to have 1 player, at least 1 crate, and an equal number of crates and targets. The solution length (pushing all the crates to the target using an A* solver) is at least 10 moves to avoid generating extremely simple levels but also not complex enough that the solver can't solve it (as shown in figure 2c).

For all the experiments, we are using a Moore Neighborhood with a size of 3×3 for local observation. For our local functions, we are using *Synchronous* cellular automata for our PCA, while the global function is *Asynchronous*. We ran all the evolved PCAs for 100 iterations, stopping when they reach a goal state. For mutation, we are using Claude 4.8 Opus with a maximum of 4096 tokens (to force small functions). Since we are using an LLM for mutation, we use Context Sampling mentioned in section 4.2. For evaluation, we evaluate each PCA on 30 random fixed starting inputs to avoid randomness affecting evaluation while still having enough samples to evaluate that PCA. Finally, for the evolution, we are using a population of 40 with elitism of 10% that we run for 50 generations. Selection uses tournament selection of size 7, and we are using uniform mutation with a rate of 10% (which means each function has a 10% chance of being replaced) and uniform crossover with equal probability between both parents.

We test two hyperparameters to understand their effect on the behavior of the PCA. We test the effect of different numbers of local functions ($l = \{1, 5, 10, 50\}$) and the different numbers of global functions ($g = \{0, 1, 5\}$). A total of 36 (3 (number of problems) $\times 4$ (number of local functions) $\times 3$ (number of global functions)) different experiments; for each one, we run it 5 times for stability (total runs equal to 180).

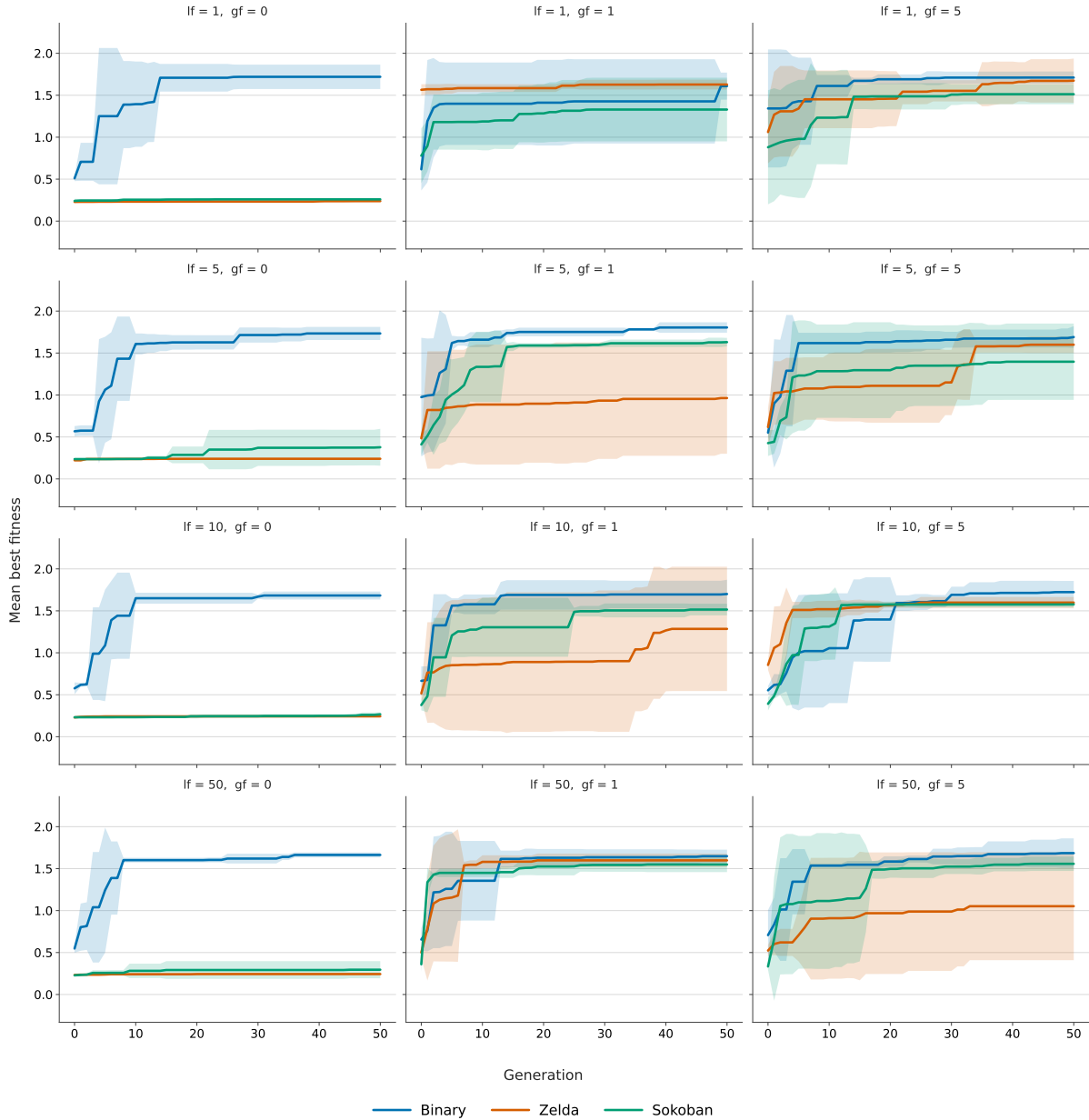


Figure 3: Fitness progression for all three games across the different configurations of the number of local functions and global functions. The error bars are the 95% confidence interval calculated over the 5 different runs.

6. Results

Figure 3 shows the fitness improvement across all three domains; we noticed that having at least one global function helps the evolution find a generator that can solve all the problems pretty efficiently. It was surprising that more global functions didn't help as much, but we believe that, due to the simplicity of the problem, as most problems are solved using 1 global function (see table 1), adding more global functions doesn't improve the evolutionary process much.

Another interesting finding, by looking at table 3, is that the average number of iterations that the PCA needs to find the best solution decreases when using global functions, except for the binary problem, where the optimum was having just 1 local function. This might be because evolution didn't optimize all the local and global functions well compared to having 1 single function. For example, if you have one global function, evolution will optimize it to be efficient at solving the problem, but if evolution has more than one function, it might split functionality or even have inefficient functions that

Game	# Global	# Local			
		1	5	10	50
Binary	0	100% $\pm$ 0%	100% $\pm$ 0%	97% $\pm$ 3%	99% $\pm$ 3%
	1	98% $\pm$ 2%	100% $\pm$ 0%	100% $\pm$ 0%	98 $\pm$ 2
	5	98% $\pm$ 2%	98% $\pm$ 1%	100% $\pm$ 0%	99% $\pm$ 1%
Zelda	0	0% $\pm$ 0%	0% $\pm$ 0%	0% $\pm$ 0%	0% $\pm$ 0%
	1	99% $\pm$ 1%	46% $\pm$ 59%	70% $\pm$ 65%	98% $\pm$ 5%
	5	93% $\pm$ 10%	100% $\pm$ 0%	99% $\pm$ 1%	67% $\pm$ 34%
Sokoban	0	2% $\pm$ 2%	16% $\pm$ 32%	0% $\pm$ 0%	11% $\pm$ 23%
	1	96% $\pm$ 5%	96% $\pm$ 3%	97% $\pm$ 5%	100% $\pm$ 0%
	5	99% $\pm$ 1%	99% $\pm$ 3%	97% $\pm$ 4%	98% $\pm$ 3%

Table 1

The playability percentage of our best generator over 100 sampled levels from each experiment with a 95% confidence interval for all three game domains.

Game	# Global	# Local			
		1	5	10	50
Binary	0	24% $\pm$ 8%	26% $\pm$ 9%	20% $\pm$ 9%	17% $\pm$ 5%
	1	16% $\pm$ 13%	33% $\pm$ 15%	23% $\pm$ 19%	14% $\pm$ 7%
	5	27% $\pm$ 9%	29% $\pm$ 12%	28% $\pm$ 12%	23% $\pm$ 19%
Zelda	0	27% $\pm$ 4%	29% $\pm$ 0%	28% $\pm$ 5%	36% $\pm$ 5%
	1	16% $\pm$ 5%	25% $\pm$ 14%	17% $\pm$ 5%	14% $\pm$ 2%
	5	22% $\pm$ 29%	11% $\pm$ 3%	14% $\pm$ 6%	22% $\pm$ 12%
Sokoban	0	2% $\pm$ 1%	3% $\pm$ 2%	1% $\pm$ 0%	4% $\pm$ 5%
	1	7% $\pm$ 2%	12% $\pm$ 5%	7% $\pm$ 4%	10% $\pm$ 4%
	5	8% $\pm$ 3%	11% $\pm$ 4%	10% $\pm$ 3%	10% $\pm$ 4%

Table 2

The diversity percentage (how many levels are different from each other) of our best generator over 100 sampled levels from each experiment with a 95% confidence interval for all three game domains.

Game	# Global	# Local			
		1	5	10	50
Binary	0	6.903 $\pm$ 0.934	9.157 $\pm$ 5.777	13.637 $\pm$ 1.032	12.350 $\pm$ 10.980
	1	15.587 $\pm$ 7.284	14.710 $\pm$ 6.425	15.593 $\pm$ 2.555	17.873 $\pm$ 14.104
	5	24.257 $\pm$ 4.803	7.420 $\pm$ 2.659	16.017 $\pm$ 13.849	15.427 $\pm$ 6.259
Zelda	0	—	—	—	—
	1	10.013 $\pm$ 1.966	58.247 $\pm$ 51.779	39.453 $\pm$ 55.971	21.920 $\pm$ 20.695
	5	28.493 $\pm$ 2.250	15.543 $\pm$ 15.337	16.240 $\pm$ 3.885	41.020 $\pm$ 28.729
Sokoban	0	98.377 $\pm$ 1.867	89.800 $\pm$ 20.091	99.673 $\pm$ 0.703	91.993 $\pm$ 17.217
	1	19.057 $\pm$ 10.468	24.223 $\pm$ 13.613	6.863 $\pm$ 4.806	13.533 $\pm$ 2.588
	5	19.853 $\pm$ 15.533	22.063 $\pm$ 13.389	24.093 $\pm$ 18.942	11.730 $\pm$ 3.382

Table 3

The number of iterations for our best generator to find a playable level over 100 sampled levels from each experiment with a 95% confidence interval for all three game domains.

can solve the same problem. Having inefficient functions might be the cause of having more iterations compared to having 1 global function.

To explain why having a global function helps in decreasing the number of iterations, imagine you are trying to count the number of player tiles in the game. To do it with local interaction, the local function and execute need to propagate whenever it finds a player tile to the rest of the grid to communicate that there is one in that location. This usually takes at least a number of iterations equal to the board size. This can be seen in work by Earle et al. [31, 54], where the NCA needs to propagate

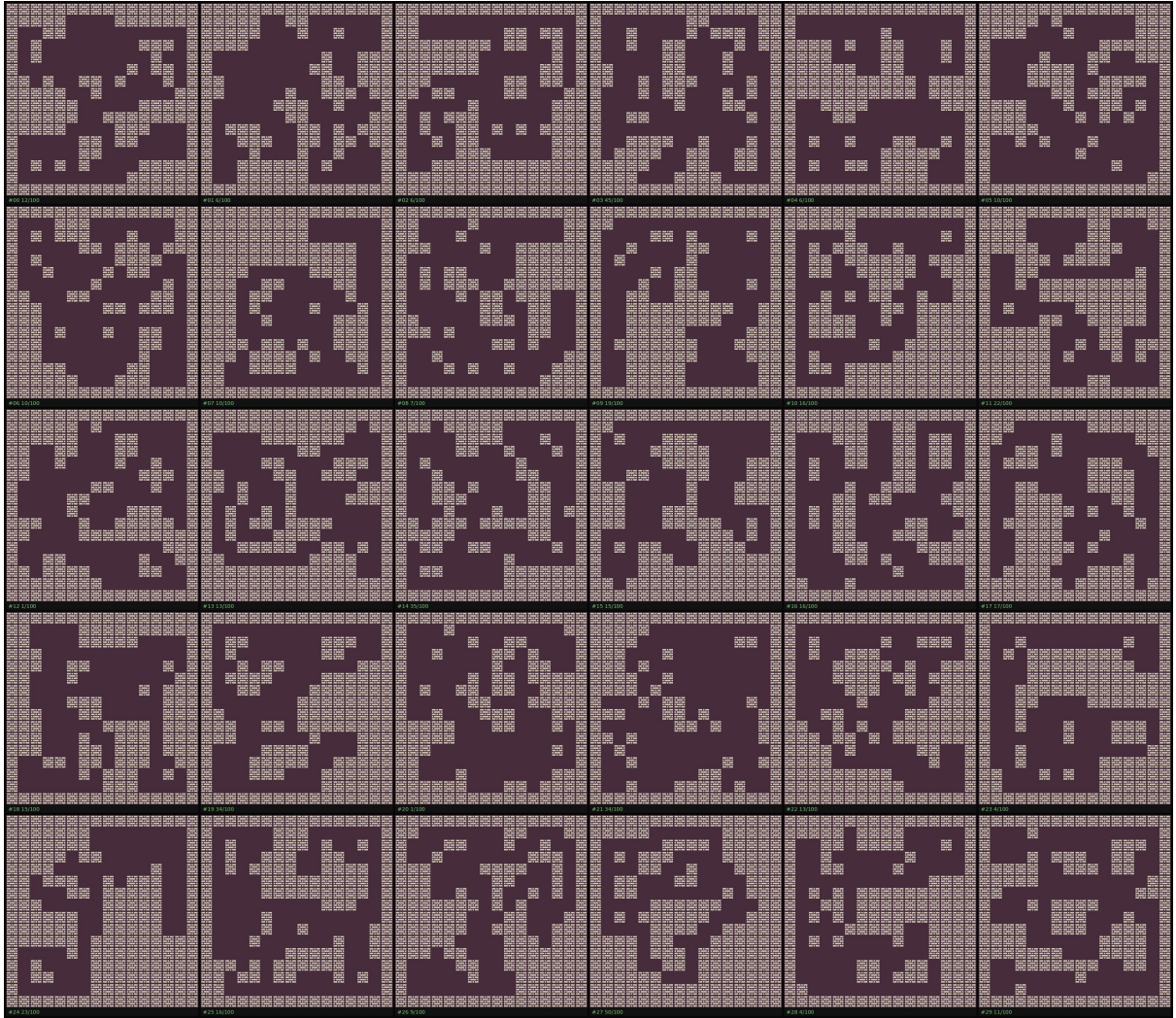


Figure 4: 30 generated levels of the Binary domain from our highest-fitness generator, which has 1 local function and 1 global function. The caption states the # of iterations per generation.

the information through the board for multiple iterations to compute some global qualities such as path length or connectivity using local computations. Having an easy way to compute these values instead of propagating a lot of information cuts the number of iterations overall. This can be easily seen in Binary since it is an easy problem and can be solved in the allocated time slots.

We analyze the best generator per domain. Table 2 shows the percentage of levels that the PCG Benchmark labels as passing the diversity criteria. A noticeable result is that although the diversity fitness is high because the fitness (in some cases almost 1.7), the number of unique levels from each other is low (all under 30%). This might be due to how we framed the diversity as part of the fitness function, where we use the distance criteria and not the percentage of different levels.

We look deeper into the generated content by analyzing the output of the fittest generator from all the runs for all three games, which can be seen in figures 4, 5, and 6. The Binary and Sokoban levels look visually diverse and have different solutions. But the Zelda levels don't look as diverse, as the PCA ended up clearing the layout and just moving the player, key, and door in random locations. This is not an issue of the evolved PCA but more of the diversity metric that is being used for Zelda. In Zelda, the diversity is measured based on the shortest path from the player towards the key and from the key to the door, so if you randomize the location of these items, you end up having good diversity.

Looking at the global functions discovered during all these runs (table 4), we notice that the most common discovered function is a count function. This makes sense since most of these games have constraints on the number of objects, which is an easy global function to discover but very hard to

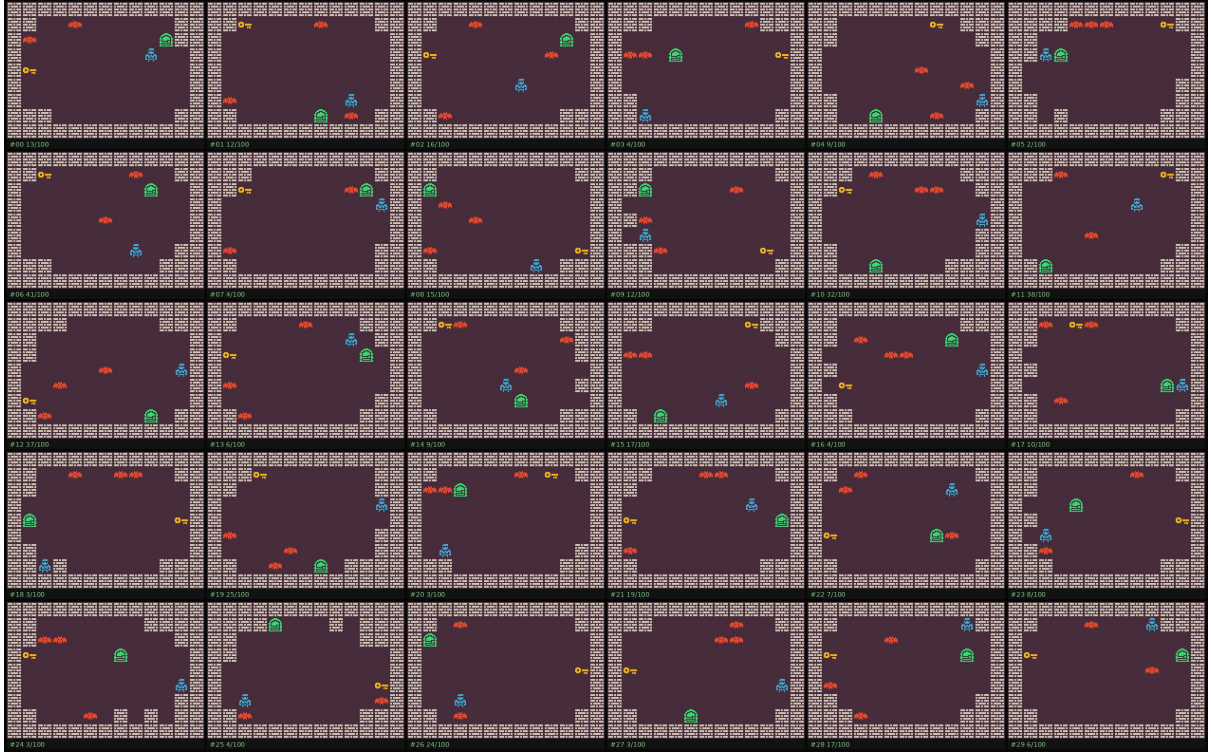


Figure 5: 30 generated levels of the Zelda domain from our highest-fitness generator, which has 50 local functions and 1 global function. The caption states the # of iterations per generation.

compute using just local functions. We can see that as soon as we have that 1 function in all the domains, it manages to solve the issue (table 1). The second most common function is about connectivity; this is also expected since in most of these games, you want the whole map to be fully connected, as if there are any isolated areas, the level is not considered playable. Finally, the other functions are about the distribution of tiles in the levels; we believe that they use these functions for aesthetics and to make sure the level is well distributed and not clustered in one location.

Table 5 shows the top 10 discovered local functions. The count function is the most common; having this count function transforms the cellular automata into an indirect type, where the decision function decides based on how many tiles surround the center tile. This is pretty common for generating layouts for levels, for example in the work by Johnson et al. [9]. We have a big group of pattern-based functions (Orientation Function, Alone Function, Vertical Function, Pattern Function, Corner Function, and Contrast Function); we believe these functions are useful for connectivity and making sure objects are spread well across the map, and the maps are fully connected (that is why they appear with a high percentage in the binary domain). Finally, other functions encode a lot of information that can be used in a full table later to make decisions.

7. Discussion & Conclusion

In this work, we explored the idea of modularizing cellular automata by introducing the idea of programmable cellular automata where each module is considered a function that can be represented by a program. We explored the usage of LLMs to generate these programs and explored the effect of their number on the output results. We expanded the formulation of cellular automata by adding global functions. Although adding global functions is not in the spirit of being fully local, the advantages of adding even 1 asynchronous global function are more than the disadvantages. Adding at least 1 asynchronous global function decreases the number of iterations the cellular automata needs to find the best output, and in some games, it helps the fitness to reach playable levels in a reasonable amount

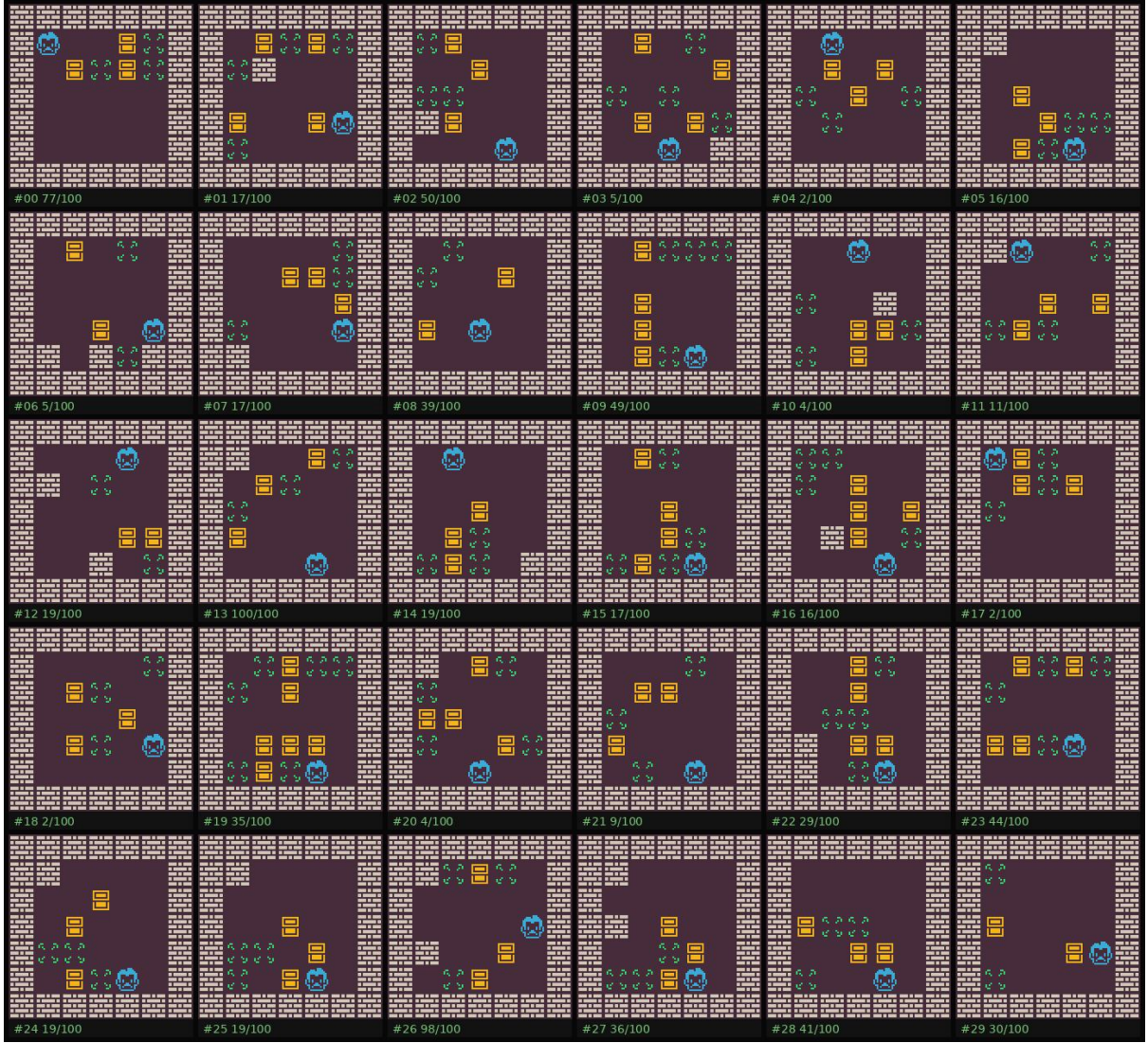


Figure 6: 30 generated levels of the Sokoban domain from our highest-fitness generator, which has 10 local functions and 5 global functions. The caption states the # of iterations per generation.

of evolution time.

In our work, the changes are visible in the current state, which means you can always see what the decision function changed, which allows for interesting movement behaviors and patterns [1]. We could expand the idea of state by having a visible part and a hidden part, where the hidden part can act like a memory from the current state to the next state. Doing that won't allow a lot of random changes in the state, and we can make sure the changes are more human-understandable, similar to the hidden states introduced in NCA [10]. Due to the scope of the paper, this was not tested in the current paper, but this extension might help the local function perform better, as they have more channels to propagate information.

Another idea is to allow global functions and local functions to return more information. This might help the overall system create more complex functions; for example, the global function can return something like a Dijkstra map, but on the other hand, it can cheat and return the final values, and the decision function just copies it.

Looking at the generated levels, we notice that a lot of the generated levels look visually similar. For example, in Zelda, a generator that generates an empty level and shuffles the position of the objects is enough to pass quality criteria. So instead of fixing the fitness function to incorporate visual diversity, we should use quality-diversity approaches similar to Earle et al.'s work [31] to explore the space of

Name	Description	# Lines	Binary	Zelda	Sokoban
Counting Function	Counts the total number of cells for a given tile.	5 ± 3	61%	61%	43%
Connectivity Function	Calculates the size of the largest connected area for a given tile.	23 ± 4	17%	4%	19%
Border Detector	Counts the number of cells on the borders of the map for a given tile.	9 ± 3	0%	6%	10%
Spread Function	Computes the perimeter of the bounding box enclosing all cells of a given tile.	11 ± 4	3%	6%	8%
Row Distribution Function	Measures how unevenly the tile type is spread across rows.	10 ± 4	0%	2%	11%
Spatial Distribution Function	Rewards tiles on the edge or the center of the map while penalizing corners.	13 ± 4	0%	3%	7%
Clustering Function	Measures how densely clustered the tile type is across the map.	15 ± 8	3%	0%	5%
Distance Function	Computes the maximum Manhattan distance of any pair of tiles in the map.	12 ± 4	0%	2%	2%
Perimeter Function	Counts the total number of edges of tiles that touch either an inactive cell or the map boundary.	15 ± 3	6%	0%	4%
Run-Length Function	Returns the maximum horizontal length of connected cells in any row.	15 ± 4	8%	1%	2%

Table 4

The percentage of time that the top 10 discovered global functions appear in all the experiments and runs (total of 40 runs per game) and the average number of lines for each of these functions.

different generators that generate visually distinct levels.

One of the limitations of our implementation is that the local and global functions don’t take all the tiles but are only applied to one tile type at a time. This limits the functions; for example, you can’t write a BFS algorithm that measures the distance between two different tile types, but you can write functions that measure distance between the same tile type. We restricted the functions to that to make sure the functions are simple, but it also limits the functions that can be produced. It would be interesting to see what will happen when we enable all tile types for all the functions.

An avenue to explore later is to see the effect of the LLM output size; in this work, we allowed the LLM to write code of any size up to 4096 tokens, which is a big size code. Although none of the generated functions were big (as shown in tables 4 and 5), it would be interesting to explore what will happen if we limit the output token size to something small, such as 128 tokens from the beginning of the evolution. This might force the evolution to be smart and find a better and more efficient solution for these functions overall. We leave this avenue to be explored in future work.

Declaration on Generative AI

During the preparation of this work, the author(s) used Grammarly in order to check grammar and spelling. We also used Claude Opus to sift through all the generated functions and give them names and descriptions for table 4 and 5. Finally, Gemini and Claude were used to find missing citations and references in the whole document.

Name	Description	# Lines	Binary	Zelda	Sokoban
Counting Function	Counts the number of active cells in the local neighborhood.	5 ± 2	100%	85%	66%
Alone Function	Returns whether the center cell is alone with no neighbors.	10 ± 4	3%	24%	46%
Orientation Function	Detect horizontal vs vertical alignment of the active cells in the neighborhood.	8 ± 3	36%	14%	36%
Encoding Function	Encodes the center cell's value with the count of active neighboring cells.	7 ± 3	33%	17%	27%
Vertical Function	Compute the vertical gradient of the neighborhood.	9 ± 3	14%	14%	26%
Weighting Function	Computes a weighted score of the neighborhood: orthogonal neighbors (20), other neighbors (7), and the center cell (1).	8 ± 2	3%	6%	31%
Pattern Function	It compares corner versus edge occupancy, then hashes that difference.	11 ± 3	0%	6%	24%
Corner Function	Computes weighted counts of corner versus edge neighbors.	10 ± 3	0%	5%	24%
Hashing Function	Computes a hashed scalar for the corner count, center presence, and edge-count parity.	7 ± 2	8%	8%	11%
Contrast Function	Compares the count of set corner cells against set edge cells depending on whether the center cell is set.	8 ± 1	0%	2%	19%

Table 5

The percentage of time that the top 10 discovered local functions appear in all the experiments and runs (total of 60 runs per game) and the average number of lines for each of these functions.

References

- [1] J. Conway, Conway's game of life, *Scientific American* 223 (1970) 120–123.
- [2] S. Wolfram, *A New Kind of Science*, Wolfram Media, 2002.
- [3] G. Egan, *Permutation city*, Greg Egan, 1994.
- [4] G. B. Ermentrout, L. Edelstein-Keshet, Cellular automata approaches to biological modeling, *Journal of theoretical Biology* 160 (1993) 97–133.
- [5] L. Kier, C. Cheng, P. Seybold, Cellular automata models of chemical systems, *SAR and QSAR in Environmental Research* 11 (2000) 79–102.
- [6] K. Nagel, M. Schreckenberg, A cellular automaton model for freeway traffic, *Journal de physique I* 2 (1992) 2221–2229.
- [7] Z. Ren, Z. Deng, Z. Sun, Cellular automaton modeling of computer network, *Computer physics communications* 144 (2002) 243–251.
- [8] W. Wright, *Lessons in game design*, 2003. URL: <https://www.youtube.com/watch?v=CdgQyq3hEPo>.
- [9] L. Johnson, G. N. Yannakakis, J. Togelius, Cellular automata for real-time generation of infinite cave levels, in: *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 2010, pp. 1–4.
- [10] A. Mordvintsev, E. Randazzo, E. Niklasson, M. Levin, Growing neural cellular automata, *Distill* 5 (2020) e23.
- [11] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., Evaluating large language models trained on code, *arXiv preprint arXiv:2107.03374* (2021).
- [12] M. C. Green, C. Salge, J. Togelius, Organic building generation in minecraft, in: *Proceedings of the 14th International Conference on the Foundations of Digital Games*, 2019, pp. 1–7.
- [13] M. C. Green, A. Khalifa, A. Alsoughayer, D. Surana, A. Liapis, J. Togelius, Two-step constructive

- approaches for dungeon generation, in: *Proceedings of the 14th International Conference on the Foundations of Digital Games*, 2019, pp. 1–7.
- [14] E. W. Hidayat, E. N. F. Dewi, I. S. Ramadhan, Application of procedural content generation system in forming dungeon level in dungeon diver game, *Jurnal Teknik Informatika (Jutif)* 5 (2024) 873–881.
 - [15] C. S. Ng, C.-H. Chen, P. M. Sathikh, A procedural approach based on cellular automata for the generation of spatial layout designs, *International Journal of Architectural Computing* 24 (2026) 107–119.
 - [16] C. S. X. Ng, C.-H. Chen, P. M. Sathikh, Hexagonal cellular automata and graph theory for procedural spatial layout generation, *Journal of Asian Architecture and Building Engineering* (2025) 1–19.
 - [17] Y. P. Macedo, L. Chaimowicz, Improving procedural 2d map generation based on multi-layered cellular automata and hilbert curves, in: *2017 16th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, IEEE, 2017, pp. 116–125.
 - [18] Z. Wu, Y. Mao, Q. Li, Procedural game map generation using multi-leveled cellular automata by machine learning, in: *Proceedings of the 2nd International Symposium on Artificial Intelligence for Medicine Sciences*, 2021, pp. 168–172.
 - [19] M. B. Temuçin, İ. Kocabaş, K. Oğuz, Using cellular automata as a basis for procedural generation of organic cities, *European Journal of Engineering and Technology Research* 5 (2020) 116–120.
 - [20] M. Cook, S. Colton, Towards procedural generation as gameplay: Clay and tombs of tomeria, in: *Proceedings of the FDG workshop on Procedural Content Generation*, 2016.
 - [21] T. Short, T. Adams, *Procedural generation in game design*, CRC Press, 2017.
 - [22] M. Mitchell, J. P. Crutchfield, R. Das, et al., Evolving cellular automata with genetic algorithms: A review of recent work, in: *Proceedings of the First international conference on evolutionary computation and its applications (EvCA'96)*, volume 8, Moscow, 1996, pp. 23–30.
 - [23] M. Mitchell, J. P. Crutchfield, P. T. Hraber, Evolving cellular automata to perform computations: Mechanisms and impediments, *Physica D: Nonlinear Phenomena* 75 (1994) 361–391.
 - [24] N. Shaker, A. Liapis, J. Togelius, R. Lopes, R. Bidarra, Constructive generation methods for dungeons and levels, in: *Procedural Content Generation in Games*, Springer, 2016, pp. 31–55.
 - [25] J. Togelius, G. N. Yannakakis, K. O. Stanley, C. Browne, Search-based procedural content generation: A taxonomy and survey, *IEEE Transactions on Computational Intelligence and AI in Games* 3 (2011) 172–186.
 - [26] T. Mahlmann, J. Togelius, G. N. Yannakakis, Spicing up map generation, in: *European Conference on the Applications of Evolutionary Computation*, Springer, 2012, pp. 224–233.
 - [27] C. Adams, H. Parekh, S. J. Louis, Procedural level design using an interactive cellular automata genetic algorithm, in: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2017, pp. 85–86.
 - [28] A. Pech, M. Masek, C.-P. Lam, P. Hingston, Game level layout generation using evolved cellular automata, *Connection Science* 28 (2016) 63–82.
 - [29] D. Ashlock, C. McGuinness, Landscape automata for search based procedural content generation, in: *2013 IEEE Conference on Computational Intelligence in Games (CIG)*, IEEE, 2013, pp. 1–8.
 - [30] C. Adams, S. Louis, Procedural maze level generation with evolutionary cellular automata, in: *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, IEEE, 2017, pp. 1–8.
 - [31] S. Earle, J. Snider, M. C. Fontaine, S. Nikolaidis, J. Togelius, Illuminating diverse neural cellular automata for level generation, in: *Proceedings of the Genetic and Evolutionary Computation Conference*, 2022, pp. 68–76.
 - [32] S. Earle, J. Togelius, Autoverse: Evolving symbolic neural cellular automata environments to train player agents, in: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2025, pp. 175–178.
 - [33] S. Sudhakaran, D. Grbic, S. Li, A. Katona, E. Najjarro, C. Glanois, S. Risi, Growing 3d artefacts and functional machines with neural cellular automata, in: *Artificial Life Conference Proceedings* 33, volume 2021, MIT Press One Rogers Street, Cambridge, MA 02142-1209, USA journals-info ..., 2021, p. 108.

- [34] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., Language models are few-shot learners, *Advances in neural information processing systems* 33 (2020) 1877–1901.
- [35] R. Gallotta, G. Todd, M. Zammit, S. Earle, A. Liapis, J. Togelius, G. N. Yannakakis, Large language models and games: A survey and roadmap, *IEEE Transactions on Games* (2024).
- [36] S. Sudhakaran, M. González-Duque, M. Freiburger, C. Glanois, E. Najarro, S. Risi, Mariogpt: Open-ended text2level generation through large language models, *Advances in Neural Information Processing Systems* 36 (2023) 54213–54227.
- [37] G. Todd, S. Earle, M. U. Nasir, M. C. Green, 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.
- [38] S. Huang, Word2minecraft: Generating 3d game levels through large language models, Master’s thesis, New York University Tandon School of Engineering, 2025.
- [39] Z. Jiang, S. Earle, A. Khalifa, J. Togelius, Agentic pcg: Procedural content generation via tool-using llms, Available at SSRN 6499021 (2026).
- [40] G. Todd, A. G. Padula, M. Stephenson, É. Piette, D. J. Soemers, J. Togelius, Gavel: Generating games via evolution and language models, *Advances in Neural Information Processing Systems* 37 (2024) 110723–110745.
- [41] C. Hu, Y. Zhao, J. Liu, Game generation via large language models, in: *2024 IEEE conference on games (CoG)*, IEEE, 2024, pp. 1–4.
- [42] M. Nasir, Y. Li, S. James, J. Togelius, Mortar: Evolving mechanics for automatic game design, in: *Proceedings of the Genetic and Evolutionary Computation Conference*, 2026, pp. 310–318.
- [43] M. U. Nasir, S. James, J. Togelius, Word2world: Generating stories and worlds through large language models, *arXiv preprint arXiv:2405.06686* (2024).
- [44] B. Romera-Paredes, M. Barekatin, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi, et al., Mathematical discoveries from program search with large language models, *Nature* 625 (2024) 468–475.
- [45] J. Lehman, J. Gordon, S. Jain, K. Ndousse, C. Yeh, K. O. Stanley, Evolution through large models, in: *Handbook of evolutionary machine learning*, Springer, 2023, pp. 331–366.
- [46] T. Taylor, M. Bedau, A. Channon, D. Ackley, W. Banzhaf, G. Beslon, E. Dolson, T. Froese, S. Hickinbotham, T. Ikegami, et al., Open-ended evolution: Perspectives from the oee workshop in york, *Artificial life* 22 (2016) 408–423.
- [47] A. Novikov, N. Vű, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. Ruiz, A. Mehrabian, et al., Alphaevolve: A coding agent for scientific and algorithmic discovery, *arXiv preprint arXiv:2506.13131* (2025).
- [48] Y. J. Ma, W. Liang, G. Wang, D.-A. Huang, O. Bastani, D. Jayaraman, Y. Zhu, J. Fan, et al., Eureka: Human-level reward design via coding large language models, in: *International conference on learning Representations*, volume 2024, 2024, pp. 26516–26560.
- [49] M. Siper, A. Khalifa, J. Togelius, Procedural content generator generation via continual abstraction discovery, *arXiv preprint arXiv:2608.17947* (2026).
- [50] R. Poli, W. B. Langdon, N. F. McPhee, J. R. Koza, *A field guide to genetic programming*, Lulu. com Morrisville, 2008.
- [51] J. R. Koza, *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems*, volume 34, Stanford University, Department of Computer Science Stanford, CA, 1990.
- [52] J. Togelius, R. De Nardi, S. M. Lucas, Towards automatic personalised content creation for racing games, in: *2007 IEEE Symposium on Computational Intelligence and Games*, IEEE, 2007, pp. 252–259.
- [53] A. Khalifa, R. Gallotta, M. Barthet, A. Liapis, J. Togelius, 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*, 2025, pp. 1–12.
- [54] S. Earle, O. Yildiz, J. Togelius, C. Hegde, Pathfinding neural cellular automata, *arXiv preprint*

arXiv:2301.06820 (2023).