Multi-Objective level generator generation with Marahel

Ahmed Khalifa  
New York University  
Brooklyn, New York  
ahmed@akhilafa.com

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

ABSTRACT
This paper introduces a new system to design constructive level generators by searching the space of constructive level generators defined by Marahel language. We use NSGA-II, a multi-objective optimization algorithm, to search for generators for three different problems (Binary, Zelda, and Sokoban). We restrict the representation to a subset of Marahel language to push the evolution to find more efficient generators. The results show that the generated generators were able to achieve a good performance on most of the fitness functions over these three problems but on Zelda and Sokoban they tend to depend on the initial state than modifying the map.

CCS CONCEPTS
- Theory of computation → Evolutionary algorithms;  
- Applied computing → Computer games.

KEYWORDS
level generation, multi-objective optimization, procedural content generation, level design

1 INTRODUCTION
Designing good levels is hard, but designing good level generators is arguably harder. The requirements on a level generator vary, and in general it is expected to produce levels that not only meet certain quality criteria, but do it consistently and with a certain degree of diversity so as to not bore the player (or designer). Faced with such a design problem, the generatively minded thinker might consider solving it by creating a level generator generator.

In a search-based framework, this is in principle much harder than creating a search-based generator. If one can formulate useful quality criteria for levels, these could be used for evolving a level generator itself. In other words, the fitness function for the generator measures the quality of its generated levels as a proxy for (or measure of) the quality of the generator.

The most obvious advantage of evolving a level generator, compared to simply evolving the individual levels, is generation speed: search-based PCG is quite slow, but an evolved generator can be much faster, in particular if it is a constructive generator. In this sense, time can be invested in evolving a generator and the investment later pays off when an arbitrarily large number of new levels can be generated in very little time. Another advantage is that finding a high-quality generator able to generate levels for a particular game can help us understand the design of the game itself, as it in some sense forms an abstraction of a space of good levels for the game. This, however, requires that the generator representation is such that a human can understand the evolved generator.

This paper describes a system for evolving level generators for 2D games. The system is based on Marahel, a previously introduced language for constructive level generators. (The version used in this paper is somewhat expanded compared to the earlier published version.) Grammatical evolution, a form of genetic programming, is used to evolve Marahel programs, and these programs are then evaluated by letting them generate a number of levels and testing the levels. As there are multiple quality criteria, multiobjective evolutionary algorithm is applied within the grammatical evolution framework. This system is applied to three different level generation problems: generating long paths and connected segments in a binary tilemap, generating levels for a simple version of the Legend of Zelda dungeon system, and generating Sokoban levels.

2 BACKGROUND
Procedural content generation (PCG) is the process of creating a game content using a computer program. PCG has been used in all different aspects of games such as textures [4, 13], rules [3, 15], patterns [7, 11], etc. PCG is usually divided based on the used methods. Each method has its own advantages and disadvantages. Three main divisions are: Constructive, Search-Based, and Machine Learning. Constructive approaches [20] applies a set of rules to generate a content. These rules are designed by the game designer to follow and find a certain content. It is usually used in the game industry due to its generation speed and direct control on the generated content. Search-based approaches [24] uses a search algorithm to find the required content. These approaches are guided using a fitness function which measures how close the current content to an ideal content. These approaches are mainly used in research and not in the industry due to the longer time it need to generate content and the indirect control on the generated content. Lastly, Machine learning approaches [22] uses a machine learning technique to generate the content. Same as search based approaches, machine learning approaches are usually used in research and not
in the industry. This is due to the need of a lot of training time and sometimes a lot of training data.

2.1 Procedural Procedural Level Generation

Procedural Procedural Level Generation Generation is the problem of using a procedural generation its high complexity. A key requirement of most procedural generators is a reasonable and workable representation of the generated content. For example: Browne and Maire [3] represented board games using Game Description Language (GDL) to be able to evolve new board games like Yavalath [25]. Finding a representation for a level generator is much higher problem as the representation should be able to represent a lot of different types of generators that can produce different content.

One of the early work in this field [23] searched the parameter space of ASP programs to generate a dungeon crawler level generator that is challenging for an automated agent. Later, Kerssemakers et al. [9] designed a meta-generator for Super Mario Bros (Nintendo, 1985) and used an evolutionary algorithm to search the space for diverse content. On a similar note, Drageset et al [5] defined a meta-generator space where each generator is defined as a set of parameters for a design generator. They used an optimization algorithm to search that space and returns the best found level so far.

Cellular Automata can be considered as level generators as they can modify the input to an new output that follow certain rules. These rules can be designed in a way to generate organic like levels [8]. Ashlock [2] evolved cellular automata rules to find different generators that generate black and white maps with full connectivity. Similarly, Pech et al [17] and Adams and Louis [1] generate cellular automata rules to generate mazes with certain features.

Another way is to represent the generator in form of neural network. Earle [6] trained fractal neural networks using A2C [14] to play SimCity (Will Wright, 1989). This might not look like level generation but if you look at SimCity as city planning problem, then the agent is generating cities. Khalifa et al. [10] explored another neural network level generator similar to Earle’s work. They represented level generation as an iterative process where at each step the agent is taking an action to improve the overall level. They used reinforcement learning to train the neural network on 3 different problems with different reward function. The trained networks were able to do learn how to modify the map from random initialization state to playable level in all the 3 problems.

2.2 Marahel Framework

Marahel [12] is a constructive level generator description language\(^1\). Each Marahel script constitutes a level generator. The script defines the generator from a bottom up approach, instead of identifying the requirement of the content, you specify the steps to reach that goal. The language was introduced to help unify the different constructive technique approaches the game designer and developer uses [21]. The generated levels by Marahel can be described as 2D matrix of integer where each integer reflect a certain game entity. Marahel script consists of 5 different parts:

- **Metadata:** contains all the required information about the size of the generated map.
- **Entities:** is a list of different game entities that can be placed in the generated level.
- **Regions:** divides the full map using an algorithm (such Binary Space Partitioning [20]) into several regions where each region is a group of tiles.
- **Neighborhoods:** is a list of relative locations that the explorers can use during generating the map. Relative locations can be used to check certain areas around a certain tile similar to the Cellular Automata neighborhoods used in cave generation [8].
- **Explorers:** is the core part of the generation. Explorers visits different tiles in the map in a certain order where each visited tile can be modified using a set of input rules. The order of the visited tiles can be defined using some parameters such as “horizontal” where it visits all the tiles one by one like scan-lines. The rules consists of two parts conditions and executers. Conditions check certain constrains to apply the executers. For example, “self(empty)” checks if the current tile (“self” neighborhood) is of entity type “empty”. Executers specify what change should happen at the location and how it is applied. For example, “all(solid)” will modify a 3x3 grid (“all” neighborhood) around the current location to be all solid entity

Marahel starts by creating a \(N \times M\) map of “undefined” entities such that \(N\) and \(M\) are defined in the Metadata. Then, it divides the map into several regions using the Regions section. Finally, Marahel applies the explorers one by one where they modify the starting map to a new map.

3 METHODS

We use Grammatical Evolution [16, 19] to evolve our Marahel programs. We restricted our evolution to only evolve five different explorers. The reason is to force the evolution to find interesting small programs than allowing for big ones. We also introduce an explorer before these five to initialize the map with random tiles to allow evolution only to focus on achieving the target results. There is no regions all the explorers are applied on the full map. For neighborhoods, we fixed them to a predefined set of 18 different ones. These neighborhoods covers different configurations that can be used such as Moore neighborhood, all the Von Neumann neighborhood, diagonal neighborhood, etc. These neighborhoods have 3 different sizes 1x1, 3x3, and 5x5.

We are going to search for generators for the same three problems introduced in PCGRL Framework [10]. The goal of generation is to find a playable level.

- **Binary:** is a 2D maze and it is the simplest problem. A good level is a level where all the empty tiles are connected using Moore neighborhood and there longest shortest path increased by \(X\) from the random initialization explorer.
- **Zelda:** is a GVGAI [18] port of the dungeon system of The Legend of Zelda (Nintendo, 1986). The goal of the game is to get a key and get to the door without dying by moving monsters. A good level is a level where there is one player,
one key, one door, and a path length between player to key and key to door is at least $X$ steps.

- **Sokoban**: a port of Japanese puzzle game by the same name. The goal of the game is to push every crate on one of the target locations. A good level is a level where there is one player, number of crates equal to number of targets, and can be solved in at least $X$ steps.

The initialization explorer that we added is adjusted similar to the one used in the PCGRL framework where it is biased to have a good starting state. In Binary, the empty is equal to solid equal to 50%. In Zelda, Empty is 50%, Solid is 25%, Enemies is 10%, Player is 5%, Key is 5%, and Door is 5%. Lastly in Sokoban, Solid is 40%, Empty is 45%, Player is 5%, Crate is 5%, and Target is 5%. The reason is to have same starting point similar to the PCGRL framework which allows us to compare it. Also, these values generate levels that require small amount of changes to make it playable therefore helping the evolution.

We decided on using multi-objective Evolution instead as we found from an earlier experiment that normal objective based evolution (GA) doesn’t improve much in all the different requirements. We think the reason that our constrained search space make it not possible to optimize all these values at the same time. An increase in one value will cause a decrease in another one which can be seen in later in section 4.

### 3.1 Representation

The chromosome consists of 102 integer number each number is a value between 0 and 49. The first number identify the number of explorers used, the second number is the Tracery seed for random number, each 20 integers after that correspond to an explorer. Each explorer number direct Tracery on which expansion to take for each non-terminal.

### 3.2 Genetic Operators

We are using two genetic operators: Crossover and Mutation. The crossover operator allows for bigger meaningful changes. It can swap either the seed number, number of explorers, or one of the explorers (all the 20 numbers). On the other hand, the mutation operator do a very small change. It picks a random location from the array and replace it with another random value.

### 3.3 Fitness Functions

In this work, we want to find generators that can produce playable levels for all the three problems. The problem of using playability only as our fitness is its rough fitness landscape. All the random initialized generators for most of the problems will produce 100% unplayable levels. Having the other fitness functions allow the space to be smoother or optimized toward these ones till reach the goal.

All our fitness function are designed to reflect how close the actual value to the desired value. For example: if we want to have one player, our fitness function will be 1 if the number of player is 1 and less than one otherwise. The value is calculate using the following equation.

\[
f(x) = \begin{cases} \frac{\text{range}_{\text{min}} - x}{\text{range}_{\text{min}}} & \text{if } x < \text{range}_{\text{min}} \\ 1 & \text{if } \text{range}_{\text{min}} \leq x \leq \text{range}_{\text{max}} \\ \frac{x - \text{range}_{\text{max}}}{\text{range}_{\text{max}} - \text{range}_{\text{min}}} & \text{if } x > \text{range}_{\text{max}} \end{cases}
\]

where $x$ is the input value to be scaled, $\text{range}_{\text{min}}$ is the minimal acceptable value, $\text{range}_{\text{max}}$ is the maximum acceptable value, and $\text{max}$ is the maximum possible value. The $f(x)$ is clamped to be always between 0 and 1.

#### 3.3.1 Binary

- **Number of Regions**: the number of regions in the generated map. The goal is to have one region so $\text{range}_{\text{min}}$ equals to $\text{range}_{\text{max}}$ equals to 1 and max is 10.

- **Path Length Improvement**: the number of increase in the shortest longest path after the random initialization explorer. The goal is to have an increase of at least 20. To achieve that, $\text{range}_{\text{min}}$ is equal to 20 and $\text{range}_{\text{max}}$ is infinity.

#### 3.3.2 Zelda

- **Number of Players**: the number of player in the generated map. The goal is to have one player. To achieve that, $\text{range}_{\text{min}}$ equal to $\text{range}_{\text{max}}$ equal to 1, and max is equal to 10.

- **Number of Keys**: the number of keys in the generated map. Similar to the number of players, the goal is to have one key.

- **Number of Doors**: the number of doors in the generated map. Similar to the number of players, the goal is to have one door.
• **Number of Enemies**: the number of keys in the generated map. The goal is to have not many enemies and not too few enemies. To achieve that, the \( \text{range}_{\min} \) is 2, \( \text{range}_{\max} \) is 4, and \( \text{max} \) is equal to 10.

• **Solution Length**: the number of steps the player need to reach the key the door. The goal is to have at least 20 steps to finish the level. Similar to path length improvement, we set \( \text{range}_{\min} \) to 20 and \( \text{range}_{\max} \) to infinity.

3.3.3 Sokoban. has four fitness functions: We have four fitness functions:

• **Number of Players**: the number of player in the generated map. The goal is to have one player. To achieve that, \( \text{range}_{\min} \) equal to \( \text{range}_{\max} \) equal to 1, and \( \text{max} \) is equal to 10.

• **Number of Crates**: the number of crates in the generated map. The goal is to have not too many crates and not too few so we set \( \text{range}_{\min} \) to 2, \( \text{range}_{\max} \) to 4, and \( \text{max} \) to 10.

• **Absolute Difference**: the absolute difference between number of crates and targets in the generated map. The goal is to have number of crates equal to number of target so the level can be won. To achieve that, we set \( \text{range}_{\min} \) and \( \text{range}_{\max} \) to 0, and \( \text{max} \) to 10.

• **Solution Length**: the number of steps the player need to win a Sokoban level (all crates are on targets). The goal is to have at least 20 steps to finish the level. Similar to path length improvement, we set \( \text{range}_{\min} \) to 20 and \( \text{range}_{\max} \) to infinity.

4 RESULTS

For the evolution, we used NSGA-II algorithm. We used tournament selection of size 2, population size of 500, number of generation equal to 2000, crossover rate equal to 70\%, and mutation rate equal to 30\%. For each problem, we have different size map similar to the same sizes from PCGRL framework. For Binary, the map size is 14x14, while Zelda is 11x7, and finally Sokoban is 5x5. Since the evolved generator will always generate different levels, so the fitness value is calculate by averaging the values of 50 different generated maps.

Because of the too many fitness for Zelda and Sokoban, the Pareto Front might be missing some interesting generator. Figure 1 shows the number of Pareto fronts at each generation. At generation 2000, we can see that for Zelda and Sokoban all there is only one front (all 500 chromosomes are in it). This show that there might be more as if we found most of the interesting front ones we will have more than one like what happened in the Binary problem with 10 fronts.

4.1 Binary

As discussed before in section 3, the current representation and restrictions don’t allow us to find a generator that satisfies both fitness functions. Having a high path length leads to having more than one, while having one region leads to having less than 20 path length improvement. Figure 2 shows the Pareto front of the Binary problem after 2000 generation. The Pareto front contains 36 chromosomes out of the 500 while the rest are distributed on the other 8 fronts.

Figure 2: The Pareto front for the Binary Problem for both fitness functions.

Figure 3 shows one of the Pareto front generator that have the highest path length improvement 0.92 (an average of 18 increase) and connectivity of 0.8 (an average of 2 regions). Looking at the generator, the generator have 3 explorers. The first explorer connect the random initialized level using vertical lines of empty which leads to a fully connected level with big open space as the connection using vertical lines instead of one tile. The second explorer is another connecting one but since the first one connected everything then it won’t happen. Finally, the last explorer goes on every tile in the map and check if it is empty using 5x5 Moore neighborhood, it convert the center to solid. This last explorer is the reason for having more than 1 region but at the same time, it is what guarantees the long path as if cause a lot of diagonal solid in big open areas. Looking at the examples in figure 3, we can see the long vertical connections leaded to having a circular dungeon generator and we can see a small few disconnected areas.

4.2 Zelda

For the Zelda problem, at the last generation all the chromosomes (500) exists in one front which shows that there is more chromosomes that can exist in the front. We decided to show all the Pareto Front for the couple of the fitness function combinations. Figure 4a shows the Pareto front between number of player and number of enemies fitness functions. It is interesting to see that it is inversely proportional as it means that having more enemies means less chance of having a single player. This makes sense as having more enemies means less tiles for the player avatar.

Figure 4b shows the Pareto front between number of doors and number of keys fitness function. Interesting enough, the graph is directly proportional everywhere except near the end where we almost have fitness of 1 for number of keys. Looking at the script, we found that the script erase a lot of entities while trying to connect the isolated area from the map. This erase behavior might break the generated levels if there is few numbers of key or doors which is the case when key value approach to 1.

Figure 4c shows the Pareto front between the number of players and solution length. The solution length fitness is pretty low overall with maximum of 0.14. This doesn’t mean it is unplayable, it also could mean very short length. Looking at the figure, it is obvious when we low number of player fitness we have low solution length as you can’t play a level if you don’t have one player. On the other hand, with high number of players the solution length is bouncing between almost 0 and 0.14. This noise is due to the other two fitness


```json
{"explorers": [
{
  "type": "connect",
  "parameters": {
    "repeats": "1",
    "replace": "buffer",
    "directions": "plus",
    "entities": "empty"
  },
  "rules": [
    "self(any) -> vert(empty)"
  ]
},
{
  "type": "connect",
  "parameters": {
    "repeats": "1",
    "replace": "same",
    "directions": "plus",
    "entities": "empty"
  },
  "rules": [
    "self(out) -> plusnc(solid|empty)"
  ]
},
{
  "type": "horz",
  "parameters": {
    "repeats": "1",
    "replace": "buffer"
  },
  "rules": [
    "plusfive(empty) -> down(solid)"
  ]
]}
```

Figure 3: The evolved binary generator and several different generated examples. On the left, the evolved explorers are shown. On the right, several different examples produced by the generator. The black tiles are solid, while the white tiles are empty. This generator has number of regions fitness value equal to 0.8 and path length improvement fitness value equal to 0.92

![Figure 4](image)

Figure 4: The Pareto front for the Zelda Problem for different combination of the fitness functions.
function: number of keys and number of doors. The only way to have a solution length is to have one player, one key, and one door. This makes it a very hard fitness function to satisfy causing these low fitness values.

Figure 5 shows a Marahel Zelda generator evolved after 2000 generation. We picked this generator as it has the highest solution length fitness. Looking on the 10 generated examples, we can notice that non of them are playable but some can be fixed to be playable easily. The interesting thing about this generator example is it have small number of players and keys and doors which make it have a high chance to generate playable levels. Looking into the generator itself, we can see it is more of an eraser. It depends on the starting noise and it tries to substitute some of the tiles by solid using a noise function while moving on the path to connect entities.

4.3 Sokoban
Similar to Zelda, all the 500 chromosomes appear in the one front. Figure 6a shows the Pareto front for the four different fitness functions. Figure 6a shows the Pareto front between number of player fitness value and solution length fitness value. It is obvious that having higher number of players will cause the solution length value to increase (you can’t play a level if you don’t have one player). The solution length value is very noisy and with peak of 4.5% around 0.6 number of player value. Similar to Zelda, the low solution length fitness value due to the cascaded fitness as a level is only playable if it has one player, number of crates more than 0 and equal to number of targets. Even when all this happens, the level have a higher chance to be unplayable compared to Zelda levels as crates can start in a locked position not allowing them to move even when all the constraints are satisfied. For example: the top left level on in figure 7 satisfies all the playability constraints but you can’t win it as the crate (red tile) can’t be moved.

Figure 6b shows the Pareto front between number of crate fitness value and absolute difference fitness value. The relation is inversely proportional as having higher number of crates fitness value causes the absolute difference value decrease. This is due to having more crates will increase the risk of having more errors and not having number of crates equal to number of targets in the generated levels. Finding a generator that produces no crates and no targets is a very easy task (a generator that erase everything). This generator will always be in the front as it will always have absolute difference fitness value equal to 1 which no other generator achieved it.

Figure 7 shows the evolved Sokoban generator with 8 different generated levels using that generator. Similarly, we picked this generator as it has the highest solution length fitness value. The generator is a bit simple and it starts by trying to connect between isolated areas in the map and use noise function with some more constraints to either add empty or target tiles. Then later it visits the new isolated tiles and adding solid tiles if there is too big of empty space. This generator also depends highly on the starting level as most of these condition only valid in certain cases and not all the time.

5 CONCLUSION
This paper introduced a multi-objective optimization method to evolve constructive level generators. The generators used Marahel [12] as their space representation. We restricted the evolution to small size Marahel scripts to force the evolution to find understandable and efficient generators. The results shows that our restrictions might have caused having a Pareto front and not able to find a generator that can achieve all the fitness functions 100%. We also see that only the binary problem was able to explore most of its Pareto front while for Zelda and Sokoban the results were a subset of the full front. The final generator for the Binary problem was interesting as it resulted into these long loopy dungeons. On the other hand, the Zelda generator acted as an eraser erasing extra objects from the random initialization, while Sokoban generator just resampled the level from a different distribution that have higher chance to be playable levels.

The evolution of Marahel resulted into a more understandable generator compared to other techniques [10]. The interpretability of Marahel language is a big advantage as we can debug these generators or edit them easily. Our representation and restrictions helped the evolution to find small concise generator that can be understood easily but at the same time it was harder to search the space. This can be noticed from our fitness functions, they created a Pareto front instead of working in tandem. It would be interesting to experiment with less restricted evolution and try to see if this will change the results. We also noticed that having an initialization explorer that initialize the map before generator explorers starts, helped to find generators that react to the current initialization by erasing instead of adding (as most of the fitness functions need less number of entities than more of them). It would be interesting to remove that initialization generator and see if we can achieve different results that tries to add more entities than erasing. Another idea, we would like in the future to try to change the fitness function to be more about improvement (similar to path length improvement) instead of optimizing towards a certain value (like number of players, number of regions, etc). We think these types of fitness functions help the evolution to find more interesting layouts and levels as it doesn’t depend on the starting state. One last thing, the average operator (used to aggregate the fitness of the generated sample maps) sometimes biases the generation towards mediocre generators. We think that using different type of operator like mixmin operator (mixing the average value and the minimum value) might forces the generator to move away from these mediocre generators.

ACKNOWLEDGEMENTS
Ahmed Khalifa acknowledges the financial support from NSF grant (Award number 1717324 - "RI: Small: General Intelligence through Algorithm Invention and Selection.").

REFERENCES
[1] Chad Adams and Sushil Louis. 2017. Procedural maze level generation with evolutionary cellular automata. In 2017 IEEE Symposium Series on Computational Intelligence (SSCI). IEEE, 1–8.

[2] Daniel Ashlock. 2015. Evolvable fashion-based cellular automata for generating cavern systems. In 2015 IEEE Conference on Computational Intelligence and Games (CIG). IEEE, 306–313.
Figure 5: The evolved zelda generator and several different generated examples. On the left, the evolved explorers are shown. On the right, several different examples produced by the generator. Black tiles are solid, white tiles are empty, green tiles are player, red tiles are enemies, yellow are keys, and cyan are doors. This generator has number of players fitness value equals to 0.79, number of keys fitness value equals to 0.67, number of doors fitness value equals to 0.7, number of enemies fitness value equals to 0.59, and solution length fitness value equal to 0.14.

Figure 6: The Pareto front for the Sokoban Problem for different combination of the fitness functions.

[3] Cameron Browne and Frederic Maire. 2010. Evolutionary game design. IEEE Transactions on Computational Intelligence and AI in Games 2, 1 (2010), 1–16.
[4] Michael F Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. 2003. Wang tiles for image and texture generation. ACM Transactions on Graphics (TOG) 22, 3 (2003), 287–294.
[5] Olve Drageset, Mark HM Winands, Raluca D Gaina, and Diego Perez-Liebana. 2019. Optimising Level Generators for General Video Game AI. In 2019 IEEE Conference on Games (CoG). IEEE, 1–8.
[6] Sam Earle. 2019. Using Fractal Neural Networks to Play SimCity 1 and Conway\'s Game of Life at Variable Scales. In Experimental AI in Games Workshop.
[7] Erin J Hastings, Ratan K Guha, and Kenneth O Stanley. 2009. Evolving content in the galactic arms race video game. In 2009 IEEE Symposium on Computational Intelligence and Games. IEEE, 241–248.
[8] Lawrence Johnson, Georgios N Yannakakis, and Julian Togelius. 2010. Cellular automata for real-time generation of infinite cave levels. In Proceedings of the 2010 Workshop on Procedural Content Generation in Games. 1–4.
[9] Manuel Kerssemakers, Jeppe Tuxen, Julian Togelius, and Georgios N Yannakakis. 2012. A procedural procedural level generator generator. In 2012 IEEE Conference on Computational Intelligence and Games (CIG). IEEE, 335–341.
[10] Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. 2020. Pcgrl: Procedural content generation via reinforcement learning. arXiv preprint arXiv:2001.05212 (2020).
[11] Ahmed Khalifa, Scott Lee, Andy Nealen, and Julian Togelius. 2018. Talakat: Bullet Hell Generation through Constrained Map-Elites. In The Genetic and Evolutionary Computation Conference. ACM.
[12] Ahmed Khalifa and Julian Togelius. 2017. Marahel: A language for constructive level generation. In Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference.
[13] Lin Liang, Ce Liu, Ying-Qing Xu, Baining Guo, and Heung-Yeung Shum. 2001. Real-time texture synthesis by patch-based sampling. ACM Transactions on Graphics (TOG) 20, 3 (2001), 127–150.
[14] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In International conference on machine learning. 1928–1937.
[15] Thorbjørn S Nielsen, Gabriella AB Barros, Julian Togelius, and Mark J Nelson. 2015. Towards generating arcade game rules with VGDL. In 2015 IEEE Conference.
Figure 7: The evolved Sokoban generator and several different generated examples. On the left, the evolved explorers are shown. On the right, several different examples produced by the generator. Black tiles are solid, white tiles are empty, green tiles are player, red tiles are crates, and blue tiles are targets. This generator has number of players fitness value equals to 0.69, number of crates fitness value equals to 0.66, absolute difference fitness value equal to 0.87, and solution length fitness value equal to 0.045.