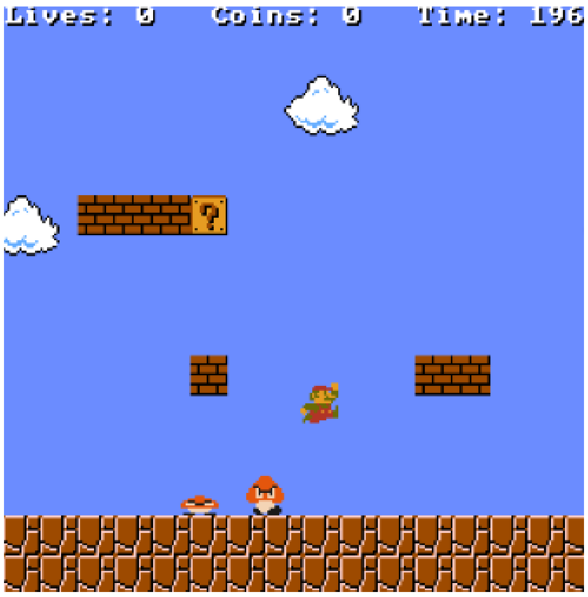
ECS7002P - Lab Exercises
MarioAI: A Level Generation Framework

11th November 2019

Mario AI Framework

MarioAI is a framework supporting game-playing and level generation for a research version of the classic video game Super Mario Bros. The framework includes several sample planning agents and level generators, as well as thousands of levels (including the original Mario levels). It is compatible with Video Game Level Corpus (VGLC) processed notations [7].



Getting started

In order to get the framework installed and running, follow these instructions:

1. Clone the following repository using git in a Windows or Unix terminal:

   git clone `https://github.com/gaigresearch/ecs7002p-marioai`

   You can also download the repository or check it out with SVN.

2. Set up your IDE to use the project. These instructions allow you to set it up with IntelliJ IDEA.
   (a) Open IntelliJ IDEA.
   (b) Select 'Create New Project' and select 'Java'. Progress clicking on 'Next'.
   (c) Type in any project name you want and select, as project location, the 'ECS7002P-MarioAI' folder (the one containing a *src/* directory). Click on Finish.
   (d) (Optional) If you already have a project open in IntelliJ, use this method instead: select 'New' → 'Create New Project From Existing Sources ...' and select the location of the 'ECS7002P-MarioAI' project. Progress clicking on 'Next'.

3. Verify that MarioAI can execute in the IDE:
   (a) Open the file PlayLevel.java.
   (b) Click on menu Run → Run, and select 'PlayLevel' in the pop up window. Alternatively, right click the file PlayLevel.java and select the option 'Run'.
   (c) It will take a few seconds for the project to be compiled and then it will run a game with an AI playing.

Exercises

To get a better understanding of the framework, have a look at the description here, which includes an extensive list of research papers:

`https://github.com/gaigresearch/ecs7002p-marioai/blob/master/README.md`

Next, this document contains more exercises for you to get familiarised with the framework. These are not assessed, but the advice is that you go through them before starting your work on the main assignment, as they should directly help your submission. You are welcome to attempt all of the exercises listed here, in the order you prefer. They are separated by suggested lab sessions, as follows:

- Lab Session 1 will encourage you to play the game yourself and understand the rules and different objects in a game. It will also allow you to run AI agents on preset or generated levels. You will then explore the Mario level search space and sample generators to get used to the framework.
- Lab Session 2 will focus on techniques for analyzing generated levels, as well as the implementation of more advanced algorithms for generating levels, or for evolving either the generators (and the quality of the levels they create) or the levels directly.
- Lab Sessions 3, 4 and 5 suggest milestones for your work on the assignment, so that you make full use of the time allocated for this during labs, when you can also receive more direct support and guidance.

Finally, the end of this document includes a reading list with papers relevant to this assignment, as well as additional material you might find useful. **Please note that additional materials are for your study only; some are not peer-reviewed (YouTube videos or blog posts), do not cite these in your assignment!**

Feel free to try any of the generator coding exercises in either **Java** or **Python**. Levels are described in plain text, therefore you may use Python code (with the same restrictions as in Java for the purpose of the assignment) to generate level files and then play these in the Java framework. You may also set up a direct connection with a Python generator to run AI agents on generated levels automatically, as long as this works on the ITL computers.

For Python code wishing to evaluate levels with simulation-based methods, we recommend building a `jar` file from the RunLevels.java class (modified accordingly if necessary, i.e. to use a specific AI agent) and run this for evaluations by specifying the file path to the level being evaluated. See details below on further usage of this class and program arguments available. To build a `jar` file in IntelliJ:

1. Navigate to 'File' → Project Structure → Artifacts
2. Click on the '+' symbol → Jar → From modules with dependencies...
3. Select your main class, and click OK → Apply (and close window)
4. Navigate to 'Build' → Build Artifacts → Build

You should then find your `jar` in path `out/artifacts/jar_name.jar`. Test if this runs in command line by running the following command: `java -jar jar_name.jar [arguments]`.

1 Lab Session 1 - Running the game & simple generators

The structure of the framework is as follows (starting from project root):

- **img**: contains sprites used to display the game.
- **levels**: contains several level files, including the original Mario levels.
- **src/agents**: contains sample AI agents provided with the framework, able to play Mario with different abilities. These include do nothing, random and A* implementations.
- **src/engine**: all Java classes necessary to run the game.
- **src/levelGenerators**: sample level generators provided with the framework. These include random generators and parameterized constructive generators.
- **src/PlayLevel.java**: main class used to run the framework.
- **src/RunLevels.java**: class used to run batches of games.

There are two main classes that can be used to run bots on different levels, generate levels and also to play the game yourself.

1.1 Playing the game

Class **PlayLevel.java** can be used to run several games with or without visuals. Run the class as it is, and play yourself through the first original Mario level! At the end of a game you will be met with a prompt which allows you to replay the level, or quit.

If you find it a bit fast, you could specify a different value for fps (frames per second) as an argument added at the end of the function in line 37:

```
MarioResult result = game.playGame(level, 200, 0, 30);
```

To change the level being played, you can specify a different filepath in line 14. All level files can be found in the `levels/` directory.

```
String levelFile = "levels/original/lvl-1.txt";
```

You will also notice that, at the end of a game, there are several game-play statistics displayed:

```
Win Rate: 1.0
Percentage Completion: 100.0
Lives: 0.0
Coins: 1.5
Remaining Time: 11.25
Mario State: 0.0
Mushrooms: 0.0
Fire Flowers: 0.0
Total Kills: 3.0
Stomp Kills: 3.0
Fireball Kills: 0.0
Shell Kills: 0.0
Fall Kills: 0.0
Bricks: 0.0
Jumps: 15.0
Max X Jump: 217.6008758544922
Max Air Time: 20.5
Brick bump: 0.0
Question block bump: 1.5
Mario hits: 0.0
```

All of these are encapsulated into a `MarioStats` object, several of which can be merged to obtain the average of multiple runs. Try to play the first original Mario level 5 times and display your average statistics at the end. How well can you play the game?

1.2 Playing an automatically generated level

To play a game generated automatically instead, change the value of the `levelFile` variable in line 14 to **null**. This will indicate that the generator defined in line 15 should be used instead. By default, the `notch` level generator (found in the `src/levelGenerators/notch` package) will be used. Can you win on this level?

If you would like to display the entire level generated, uncomment line 30 in this file:

```
game.buildWorld(level, 1);
```

This function will open a new window (separate from your game window), which displays the entire level. The second parameter passed to this function is the scale of the level, but we recommend not changing this. You can try it with the scale set to 0.5 for a more complete (albeit smaller) view of the level generated.

Finally, you can play a different level generated by the same generator every time you restart from the GUI prompt at the end of a game, if you set the `generateDifferentLevels` flag to true (line 14).

1.3 Running AI players on levels

We've seen previously that line 37 plays the game with visuals and a human in control of Mario. However, we can let an AI agent take control of Mario instead, if we use the execution method in line 40 instead (make sure to comment line 37 to see only the AI playing):

```
MarioResult result = game.runGame(agent, level, 20, 0, visuals);
```

This function will run the agent defined in line 24 (an A* agent by default), on the given level (which can preset or automatically generated), with a limit of 20ms of thinking time per game tick, with visuals. You can turn off visuals for faster execution.

Try out different AI agents on a preset level and compare their performance. Have a brief look at some of the AI agent code; can you figure out what they do?

1.4 Running multiple games

Class **RunLevels.java** can be used similarly to run a batch of games, useful for experiments and detailed statistics. You can modify the run settings directly in the file (lines 23-27) or run the class with program arguments (if any arguments are set, these will be used instead and overwrite the default options in the file):

```
RunLevels.java usage: 4+ args expected
    [index = 0] number of levels
    [index = 1] repetitions per level
    [index = 2] using level generator (boolean; using preset levels given
       in arguments 4+ if false)
    [index = 3] AI agent thinking time (20ms default)
    [index = 4...] preset levels (if not using generator). there should be
       exactly the number specified before (index 0) of filepaths,
       separated by space.
```

For example:

```
2 2 false 20 "levels/original/lvl-1.txt" "levels/original/lvl-2.txt"
```

will run the 2 specified levels, 2 times each.

```
5 10 true 20
```

will generate 5 levels and run each one 10 times.

The AI agent and level generator for these runs are set directly in the file, in lines 48-49:

```
MarioAgent agent = new agents.robinBaumgarten.Agent();
MarioLevelGenerator generator = new levelGenerators.notch.LevelGenerator();
```

A progress indicator is printed after each game, giving the outcome of the game just played, and at the end of the game there are detailed average statistics displayed.

Run the `robinBaumgarten` and `sergeyKarakovskiy` agents several times on the same preset level and observe their performance.

1.5 Level generators

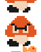
To create a Mario level generator, the Java class must implement the `src/levelGenerators/MarioLevelGenerator.java` interface. This requires 2 methods to be implemented by any generators:

- **String getGeneratedLevel(MarioLevelModel model, MarioTimer timer)**: the main function called for a generator to return a level. Levels must be in String format, with lines separated by ‘\n’ characters and containing the sprite mapping as defined in the `MarioLevelModel` object. The function must return the level in the allocated time, passed as a `MarioTimer` object.
  - The `MarioLevelModel` class may be used for storing the generated level in a format more accessible than a String object. Methods such as *setBlock(...)*, *setRectangle(...)*, *getBlock(...)* are key for managing the contents of the level, which can then be retrieved in String form using the *getMap()* method. The static methods in this class can be used to retrieve different sprite types.
- **String getGeneratorName()**: all good generators should have a name!

Analyze the information available in the `MarioLevelModel` object and consider how a generator might use the methods provided to create a level.

1.6 Level search space

The sprite mapping used in the framework is as follows:

Sprite	Character	Image
MARIO_START	’M’	
MARIO_EXIT	’F’	
EMPTY	’-’	
GROUND	’X’	
PYRAMID_BLOCK	’#’	
NORMAL_BRICK	’S’	
COIN_BRICK	’C’	
LIFE_BRICK	’L’	
SPECIAL_BRICK	’U’	
SPECIAL_QUESTION_BLOCK	’@’	
COIN_QUESTION_BLOCK	’!’	
COIN_HIDDEN_BLOCK	’2’	
LIFE_HIDDEN_BLOCK	’1’	
USED_BLOCK	’D’	
COIN	’o’	
PIPE	’t’	
PIPE_FLOWER	’T’	
BULLET_BILL	’*’	
PLATFORM_BACKGROUND	’—’	
PLATFORM	’%’	
GOOMBA	’g’	
GOOMBA_WINGED	’G’	
RED_KOOPA	’r’	
RED_KOOPA_WINGED	’R’	
GREEN_KOOPA	’k’	
GREEN_KOOPA_WINGED	’K’	
SPIKY	’y’	
SPIKY_WINGED	’Y’	

Investigate how the characters are put together in level files, the different other sprites that can be spawned, and what these look like when the entire level is displayed (Hint: use the `MarioGame.buildWorld()` method seen in previous exercises).

Modify an existing level to include different features, and play in the new level - does it feel different?

Hand-design a level from scratch with multiple of the available features. Make notes of what elements you are considering and why (e.g. many gaps to make Mario jump a lot? many different heights?). What makes a level playable? What makes a level enjoyable?

1.7 Explore sample generators

Have a look at some of the sample level generators and try to follow their logic. All provided samples are *constructive* generators: they follow certain hand-designed rules and assigned probabilities to create different features in the level and place tiles accordingly. Are you able to follow the steps they use to create levels?

Try running some different generators and observe the different levels they create. Are they all playable levels? Are they interesting? How do AI agents perform on these levels?

How could the generators be further improved?

2 Paper Read

For next week's lab, read the original MarioAI competition paper[1] that summarizes the contest and some AI approaches [3].

Additional materials suggested:

- "Towards Automatic Personalized Content Generation for Platform Games" by Shaker et al. [5].
- "Patterns and Procedural Content Generation: Revisiting Mario in World 1 Level 1" by Dahlskog and Togelius [2].
- Michael Cook's tutorial on generative and possibility spaces[2].
- Tommy Thompson's summary of the original MarioAI competition[3].

[1]`http://julian.togelius.com/Karakovskiy2012The.pdf`
[2]`http://www.possibilityspace.org/tutorial-generative-possibility-space/index.html`
[3]`http://www.aiandgames.com/the-mario-ai-competition`

3 Lab Session 2 - Advanced level generation

3.1 Parameterize the random generator

Copy the random generator into a new class (give it a new name). We'll be modifying the new class so as to leave the original code intact. Make the new class implement the `ParamMarioLevelGenerator` interface instead and add the two new methods:

```java
ArrayList<float[]> getParameterSearchSpace() {return null;}
void setParameters(int[] paramIndex) {}
```

In the first method, `getParameterSearchSpace`, we will define the parameter search space. To do this, create a new *ArrayList* and add to the list one *float* array for each of the random generator's parameters (the list should contain 8 float arrays by the end). For each of the parameters, specify a few values in its float array that that parameter can take. You can use in-line array creation to keep the code readable; for example, for the first parameter (ground Y location) we can write the following to allow 2 values:

```java
ArrayList<float[]> searchSpace = new ArrayList();
searchSpace.add(new float[]{13, 14});  // Ground Y location
```

Continue similarly for the other 7 parameters and return the *searchSpace* variable at the end.

In the second method, `setParameters`, we assume we receive an array with an index for each parameter in the search space previously defined (e.g. if $paramIndex[0] = 0$, that means the first parameter's index is 0, which corresponds to the value 13 in our search space defined above, or the first element in the float array for the first parameter). All this method needs to do is retrieve the search space, and assign to each parameter its corresponding value as indicated in the *paramIndex* array.

Now we can investigate the search space of this generator from outside the class, and also run it with different parameter configurations by setting its parameters before a run! Try to run the generator with different parameters, set in the PlayLevel.java class (and not directly modified in the random generator class).

3.2 Level evaluation

Similarly to how we use heuristics or value functions to evaluate non-terminal game states in search algorithms, in order to be able to automatically generate levels, we need an idea of *how good* a particular level is. The next two exercises will focus on evaluating levels and assigning *fitness* values to different levels.

Several evaluation methods may be used in generating levels:

- None: no evaluation necessary in the method, typical of constructive methods.
- Feature-based: design a set of requirements the generated level should meet and evaluate levels based on how closely they match the criteria.
- Simulation-based: use AI agents included with the framework to evaluate levels and measure their gameplay against a set of desired criteria.
- Human-in-the-loop: humans plays levels and give feedback, based on which the level generation improves over time.

We will explore the two in the middle next.

3.2.1 Feature analysis

First off, let's define several features for the levels and analyse them according to a pre-defined concept of *good levels*. Create a new function in class `PlayLevel.java` which will receive as an argument the String of a level and will output a *double* value. In the body of this function:

1. Identify several features which can tell you if a level is good or not, for example:
   - ϕ_1 Number of gaps in the floor
   - ϕ_2 Proportion of ground tiles to floating tiles
   - ϕ_3 Density of enemies of different types

2. Calculate the value of this features for the given level string (e.g. $\phi_1 = 8$). You might find it easier to first build the level in a MarioWorld object: for this, have a look at function `MarioGame.buildWorld(String level, float scale)`.

3. For each of the features, define an *ideal* value (e.g. the perfect level has $\phi_1^* = 5$). Calculate the difference between the actual and ideal values for each feature (e.g. $\Delta_1 = 3$).

4. For each of the features, define a weight giving the importance of the feature (e.g. $w_1 = 0.3$) where all weights sum up to 1.

5. Calculate the fitness of the level by summing all the feature differences (Δ_i) multiplied by their corresponding weights (w_i) and return this value.

In the main method of the class `PlayLevel.java`, set the *generateDifferentLevels* variable to true and, within the WHILE loop, include an evaluation of all the levels being played using your new function and print out the fitness values. Run the class and play several levels, paying attention to the fitness values printed.

Does this quantified value match your idea of the level quality while playing it? How does the average fitness per level value compare between different generators?

3.2.2 Simulation analysis

Often times, just looking at the features of a level might not give a complete picture of what the player experience might be while actually playing the game. As human evaluations can be expensive, let's use AI agents to play the game and base our judgement of which levels are good on the performance of the agents.

1. Run several agents (e.g. AStar, random) on the same level (e.g. in class `PlayLevel.java`).

2. Let's assume that a *good* level in this case would be one in which smarter agents perform better, and less intelligent agents perform worse, with successful random play being discouraged.

3. Given the win rate of the agents on a level, can you express this *good* level goal in an equation?

4. Compute a fitness value for the level based on your equation and print the resulting value.

Play the levels yourself after the AI evaluation and inspect the new fitness values. Do these match your own experience better or worse than in the previous exercise? How do different generators compare?

Could you identify other features that could be extracted from an AI player's gameplay data to better inform level fitness? (i.e. measure player experience rather than player performance)

3.3 Evolve! - Search-based generation

Now that we have a way to evaluate levels automatically according to a definition of *good* levels, we can perform a search in the space of possible levels to find ones that most closely match our definition.

Therefore, you can use the previously defined functions to evolve:

- The parameters of the parameterized random generator (Exercise 2), so that it produces better levels over several generations. The search here happens in the random generator parameter search space (i.e. evolving things that make levels).
- An initial level (random, generated, human-designed) so that it becomes better over several generations. The search here happens in the level expression search space (i.e. evolving the levels themselves).

You can use the sample evolution code for a Random Mutation Hill Climber provided on QMplus to build these two algorithms. They should be starting from an initial solution (vector of parameters, or level string), evaluate this solution using your evaluation functions, then use evolutionary concepts to create better levels over time. In this exercise, try to evolve your levels for 100 generations if you're using a feature-based analysis, or 20 generations if you're using a simulation-based analysis.

Are the levels evolved increasing in fitness over generations? Are they actually improving according to your subjective definition of *good* levels? How could you make it so that the quantitative measures of level quality better match up to your expectations? How could you create *interesting* or *different* levels instead?

3.4 Algorithmic generation

Create a new Java class in a package inside `src/levelGenerators` which implements the `MarioLevelGenerator` interface. In this file, add a default constructor for your generator which instantiates a random number generator to be used in the class. Then add the 2 methods required by the interface (leave them blank for now).

In the `getGeneratedLevel` method, implement the concepts of a Cellular Automata algorithm, following several steps:

1. Get the list of tiles to be used in the level (use static methods from the `MarioLevelModel.java` class, or create your own selection from the tiles described in that class). Do not include Mario or the exit in this list.
2. Randomly initialize a level matrix with all possible sprites. You may want to use different probabilities for the different tile types.
3. Until timer runs out, keep looping over all the tiles in the model and update them according to some rules defined in a separate function, which should follow this logic:
   (a) Get character at current position and its 8 neighbours.
   (b) Count how many tiles of each type show up in the neighbour list.
   (c) Apply transformation rules based on current tile and neighbours (i.e. if the neighbours have less than 3 *EMPTY* tiles, the current tile will also become *EMPTY*).
4. At the end of the loop, place Mario in the beginning of the level and the exit at the end.

Try to run your generator (i.e. in the `PlayLevel.java` class) and check what kind of levels it makes. Are they playable? How could you ensure the generated levels *are* playable, if yours currently aren't? How could you improve this generator further? Can you visualize the levels at different stages in the updates?

4 Paper Read

For next week's lab, read about evolving Mario levels in the latent space of a Deep Convolutional Generative Adversarial Network [8].

Additional materials suggested:

- "Experiments in Map Generation Using Markov Chains" by Snodgrass et al. [6].
- "A Procedural Procedural Level Generator Generator" by Kerssemakers et al. [4].
- Michael Cook's tutorial on sampling levels to evaluate generators[4].
- Tommy Thompson's review of Super Mario Bros. level design research[5].
- Tommy Thompson's tutorial on building Mario levels with Machine Learning[6].

[4]`http://www.possibilityspace.org/tutorial-sampling`
[5]`http://www.aiandgames.com/the-legacy-of-super-mario-bros/`
[6]`http://www.aiandgames.com/mario-makers`

5 Lab Session 3 - Assignment work: planning

This and the following two sections are for your guidance only and suggestions for your assignment work milestones, in order to make full use of lab time and support available.

In the third lab for this part of the module you should start working on your assignment. For this lab session you should have already read the recommended reading and we suggest you start experimenting with different options (i.e. what combination of methods would make sense / be easy enough to implement / be challenging enough etc.) and come up with a plan of what you're going to do, specifically:

- **Algorithm(s) to implement.** Find resources to help you with this and make sure you have a clear idea of how they work. How long is this expected to take? What if the implementation of your chosen method is more difficult than you'd anticipated, any backup plans?
- **Content representation.** How are you going to represent your levels so that it is easiest to work with for both your chosen algorithm(s) and the framework?
- **Evaluation method(s).** What method are you going to use? Design rules or measures for your generated levels.
- **Experimental process.** How many iterations are you going to run for your algorithm, how long is the generation expected to take, etc.

Talk to your group and make sure you're all on the same page, and delegate tasks so that everyone has something to do.

Write up the plan you've decided in your report (**introduction**, **methods** and **experimental setup** sections can already be drafted, even if just as bullet points!).

6 Lab Session 4 - Assignment work: implementation

In the fourth Mario AI lab you should start implementing your methods and check if your plan is actually achievable, or make changes accordingly. Again, split the tasks between group members, for example:

- One person could be implementing algorithms.
- One person could be implementing evaluation methods.
- One person could be setting up the experimental setup into which all methods should be easily plugged in and output information easy to interpret/analyse later on.

Or you could practice pair programming!

Update the **methods** and **experiments** sections in your report to better reflect your actual implementation. Fill in the **background** and **literature review** information which would support your implementation and experimental process.

7 Lab Session 5 - Assignment work: analysis + playtest!

In the last lab you should have methods implemented and some results ready for analysis! Write up analysis and plotting software (e.g. fitness plots for evaluation algorithms, visualizations of different levels produced by your generators etc.), which would produce images and graphs to be used in the report.

You might also find it useful and interesting for your assignments to have peers playtest the levels generated by your algorithm and give feedback and qualitative evaluations. Define several questions (your own interests or a validated questionnaire, e.g. [1]) that you would like to ask your teammates and invite your peers to play some levels and answer your questions! You can use an analysis of the questionnaire results in your assignment.

Write up the **results** of your experiments, their **analysis** and **conclusions** in the report. You should now have a final draft of the report and code, to be polished (formatting, commenting code etc.) for the deadline!

References

[1] Ahmad Azadvar and Alessandro Canossa. Upeq: ubisoft perceived experience questionnaire: a self-determination evaluation tool for video games. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, page 5. ACM, 2018.

[2] Steve Dahlskog and Julian Togelius. Patterns and Procedural Content Generation: Revisiting Mario in World 1 Level 1. In *Proceedings of the First Workshop on Design Patterns in Games*, page 1. ACM, 2012.

[3] Sergey Karakovskiy and Julian Togelius. The Mario AI Benchmark and Competitions. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):55–67, 2012.

[4] Manuel Kerssemakers, Jeppe Tuxen, Julian Togelius, and Georgios N Yannakakis. A Procedural Procedural Level Generator Generator. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 335–341. IEEE, 2012.

[5] Noor Shaker, Georgios Yannakakis, and Julian Togelius. Towards Automatic Personalized Content Generation for Platform Games. In *Sixth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2010.

[6] Sam Snodgrass and Santiago Ontañón. Experiments in Map Generation Using Markov Chains. In *FDG*, 2014.

[7] Adam Summerville, Sam Snodgrass, Michael Mateas, and Santiago Ontanon. The VGLC: The Video Game Level Corpus. *Proceedings of the 7th Workshop on Procedural Content Generation*, 2016.

[8] Vanessa Volz, Jacob Schrum, Jialin Liu, Simon M Lucas, Adam Smith, and Sebastian Risi. Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 221–228. ACM, 2018.