



**FACULTY
OF MATHEMATICS
AND PHYSICS**
Charles University

MASTER THESIS

Július Flimmel

**Coevolution of AI and level generation
for Super Mario game**

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Vojtěch Černý

Study programme: Computer Science

Study branch: Computer Graphics and Game
Development

Prague 2020

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague, 28 July 2020

.....

Author's signature

I would like to thank my whole family, girlfriend and friends for their support throughout my studies at the university. I would also like to express my sincere gratitude to my supervisor of this work, Vojtěch Černý, for his guidance, help and the time he spent with it. Last but not least, I would like to thank all the great teachers at the Faculty of Mathematics and Physics which made the studies more interesting.

Title: Coevolution of AI and level generation for Super Mario game

Author: Július Flimmel

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Vojtěch Černý, Department of Software and Computer Science Education

Abstract: Procedural Content Generation is now used in many games to generate a wide variety of content. It often uses players controlled by Artificial Intelligence for its evaluation. PCG content can also be used when training AI players to achieve better generalization. In both of these fields, evolutionary algorithms are employed, but they are rarely used together. In this thesis, we use the coevolution of AI players and level generators for platformer game Super Mario. Coevolution's benefit is, that the AI players are evaluated by adapting level generators, and vice versa, level generators are evaluated by adapting AI players. This approach has two results. The first one is a creation of multiple level generators, each generating levels of gradually increased difficulty. Levels generated using a sequence of these generators also mirror the learning curve of the AI player. This can be useful also for human players playing the game for the first time. The second result is an AI player, which was evolved on gradually more difficult levels. Making it learn progressively may yield better results. Using the coevolution also doesn't require any training data set.

Keywords: coevolution, artificial intelligence, procedural content generation, Super Mario

Contents

Introduction	4
Goals of this thesis	5
Structure of this thesis	5
1 Background	6
1.1 Platformer games	6
1.1.1 Game levels	6
1.1.2 Super Mario Bros game	7
1.1.3 Mario AI Framework	9
1.2 Procedural Content Generation in games	9
1.3 Artificial Intelligence in games	10
1.4 Artificial neural networks	10
1.4.1 Neuron	10
1.4.2 Feedforward layered networks	11
1.4.3 Recurrent networks	12
1.4.4 Artificial neural networks in games	12
1.5 Evolutionary algorithms	12
1.5.1 Genetic algorithms	13
1.5.2 Neuroevolution	14
1.5.3 Coevolution	16
1.6 Markov chains	18
2 Analysis	19
2.1 AI player evolution	19
2.2 Level generator evolution	20
2.3 Coevolution	20
2.4 Game engine	21
2.5 Kotlin	21
2.6 Third party libraries	21
3 AI player evolution	24
3.1 Related works	24
3.2 Our approach	24
3.2.1 Fitness function	26
3.2.2 Training data set	26
3.2.3 Neuroevolution of weights	27
3.2.4 NEAT	27
3.3 Our results	27
3.3.1 Experiments: Neuroevolution of weights	27
3.3.2 Experiments: NEAT	29
3.3.3 Additional improvements	29
3.3.4 Running times	31
3.4 Conclusion	31

4	Level generators evolution	34
4.1	Related works	34
4.2	Our approach	35
4.2.1	Chunked Markov Chain level generator	35
4.2.2	Multipass level generator	37
4.2.3	Level postprocessing	40
4.2.4	Evolution	40
4.2.5	Fitness function	41
4.3	Our results	44
4.4	Conclusion	48
5	Coevolution of AI and level generators	49
5.1	Related work	49
5.2	Our approach	50
5.2.1	Encountered issues	51
5.3	Chosen algorithms	51
5.4	Our results	52
5.4.1	Experiments using neuroevolution of weights	52
5.4.2	Experiments using NEAT	54
5.4.3	Running times	54
5.5	Evaluation of AI players	54
5.6	Evaluation of level generators	54
5.7	Conclusion	58
6	Implementation	59
6.1	Code organisation	59
6.2	Mario AI Framework integration	60
6.3	Evolutions	60
6.4	Coevolution	60
6.5	Visualisations	61
6.5.1	Level visualiser	61
6.5.2	Evolution charts visualiser	61
6.6	3rd party libraries changes	62
6.6.1	NEAT library changes	62
6.6.2	Mario AI Framework changes	63
6.7	KDoc	63
	Conclusion	64
	Future work	65
	Bibliography	66
	List of Abbreviations	70
A	Attachments	71
A.1	Electronic attachment	71
A.1.1	Experiments folder	71

B	Running the evolutions	73
B.1	Gradle	73
B.1.1	Gradle wrapper	73
B.1.2	Gradle projects	73
B.1.3	Gradle tasks - MarioAI4J	73
B.1.4	Gradle tasks - MarioDoubleEvolution	73
B.2	Launchers	74

Introduction

The roots of computer games date back to the 1960s. In 1962, Steve Russell created a program, which is considered to be the first computer game, called *Spacewar!* [1, Chapter 2]. The game consisted only of two user-controlled spaceships, a star with gravity in the middle and a few small stars in the background with no functionality. Since then, computer games evolved rapidly. Today's games content is much larger; for example, in *No Man's Sky* (Hello Games, 2016) the player can visit more than 18 quintillions (2^{64}) planets [2].

Generating this amount of content is possible only by *Procedural Content Generation* (PCG). Using this technique, the content is generated algorithmically, rather than manually (described in more detail in Chapter 1). Not only it allows for a much larger amount of content in games, but it also speeds up game development and improves the game's replayability.

PCG was used in computer games since the 1980s [3]. At the time, its main benefit was a smaller game size. As technology evolved, space capacity became less of an issue, and the main advantages of PCG became its gameplay improvements. Nowadays, it is rarely used primarily to save space, but there are still games (e.g. *No Man's Sky*) whose size would probably be unsustainable without PCG. It is used to generate a large variety of content in various games, including *textures, music, vegetation, buildings, entity behavior, levels, story* and more. Some examples of famous games using PCG are *Diablo 3* (Blizzard Entertainment, 2012) for map and items generation, *Minecraft* (Mojang Studios, 2009) for world generation, *Borderlands 3* (2K Games, 2019) for weapons generation or *Spore* (Electronic Arts, 2008) for creatures and their animations generation.

When generating game maps or levels via PCG, they need to be evaluated to determine if they fit the game's needs. This can be done by letting players play through the generated content and let them measure its quality. However, when using human players, this approach requires many resources. Using players controlled by Artificial Intelligence (AI) is more suitable because they can play through the content much faster, resulting in more evaluations in a shorter time (and without subjective errors). Another advantage of AI players is that they can be used during runtime in the background. Still, there exist games that use human-controlled players' performance to adjust game's content, for example, to tailor the game experience to player's game style like in *Left for Dead* (Valve, 2008) [4].

AI was present in games since the beginnings of the game industry. Many games used it to control *non-player characters (NPCs)*. One of the earliest NPCs were *aliens* in *Space Invaders* (Taito, 1978). However, they only moved on fixed paths. One of the first NPCs which reacted to the environment appeared in *Pac-Man* (Namco, 1980). In this game, *ghosts* chased or ran away from the player, depending on the game's state. Next to controlling only characters, AI was also used to control players in games. There had been *Pong* (Atari, 1972) clones where AI controlled one of the paddles [5]. In today's games, AI players are used in many game genres: *sports games, real-time strategies, racing games, card games*, and more.

AI-controlled players can also be used in single-player games. One reason for

this is that some games are suitable for benchmarking different AI approaches. For example, Super Mario was used for multiple AI championships [6]. Another usage of AI players is for content evaluation, as already mentioned.

In this thesis, we will be generating content via PCG and evolving AI players for *Super Mario* game. The game was released in 1985 and now it is considered a classic. Since then, multiple newer versions were released and are still being released to this day. Not only that, but it also laid the foundations to the whole platformer games genre. This makes the game still relevant for research even today, since its results can be extended to multiple other games in the genre.

Goals of this thesis

In this thesis, we will use evolutionary algorithms to evolve AI players and level generators for a Super Mario game. We will implement multiple algorithms in both cases, and then we will choose the fittest ones and combine them in one coevolution. We specify two goals of the coevolution:

- Create a sequence of level generators for Super Mario game where each next generator produces slightly more challenging levels than the previous one,
- Explore whether evolving AI players using coevolution is more feasible or yields better results than using single evolution.

Structure of this thesis

The following work is divided into multiple chapters. In Chapter 1 we describe the platformer games genre and Super Mario game. Then we define and briefly explain fields relevant to this work - PCG and AI. At the end of this chapter, we give a high-level overview of the algorithms we will use. In Chapter 2 we analyze our goal and propose how we will solve it. In the following chapters 3, 4 and 5 we look at AI, level generation, and their coevolution in more detail. We mention related works and describe our work in these fields. In the final Chapter 6 we explain, how we implemented our work. It is followed by the conclusion of the thesis and a list of options for future work.

1. Background

In this chapter we will briefly describe all the fields relevant to this work. We start with describing platformer games genre and *Super Mario Bros* game. Then we continue with outlining *procedural content generation* and *artificial intelligence* in games. Finally, we will look at the algorithms used in this work. These algorithms include primarily evolutionary algorithms, neural networks, their combination using neuroevolution and coevolution. We will also describe Markov chains, which is a statistical model we will use for level generation.

1.1 Platformer games

Platformer games (or platformers) is a popular genre among computer games. The main characteristic of this genre is that the player controls a character, which needs to jump or walk over many platforms placed throughout the world while avoiding obstacles to finish the game. Probably the most common obstacles are gaps between platforms, which usually need to be jumped over. Another type of obstacles are different kinds of enemies with their custom mechanics specific for each game. Many platformers contain also some kind of collectibles, which can increase player's score, player's character's skills or have other benefits.

While platformers started as 2D games, there are also some famous 3D incarnations like *Crash Bandicoot* series (Sony Computer Entertainment, 1996-1998). The main mechanics in 3D worlds stay the same as described above.

1.1.1 Game levels

Some games are split into levels. These are discrete parts of the game with a given objective for the player. Especially platformer games are commonly divided into many levels, where each level's objective is to reach a given place in the level without losing all lives as depicted in Figure 1.1.

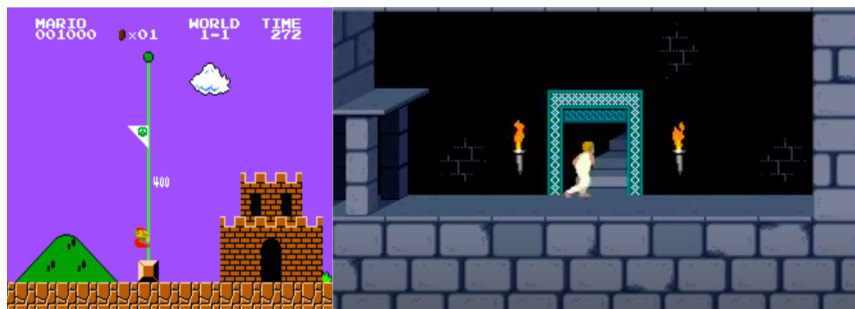


Figure 1.1: Depiction of different objectives in platformer games. On the left side we can see *Super Mario Bros* level finish, which is always on the right end of the level. On the right side we can see *Prince of Persia* level finish, which the player first needs to find and unlock.

1.1.2 Super Mario Bros game

Mario Bros game, released in 1983 by Nintendo was the first game in the Super Mario game series. Its subsequent successor, *Super Mario Bros* (Figure 1.3), released in 1985, is now a classic platformer game. The games' rules are almost the same for each game in the series, excluding the non platformer versions and 3D incarnations, and pretty simple.

In *Super Mario Bros*, the player controls its main protagonist, *Mario*, to finish multiple levels to save princess *Peach*. Mario needs to be navigated through many platforms in the levels, trying to avoid different obstacles to reach goal pole or (fake) Bowser at the end of each level. Mario can perform only a few actions, which are *move right*, *move left*, *jump* and *shoot*. Moving right and left can be combined with *sprint*, which increases Mario's speed. Shooting is available only if Mario manages to pick up a specific powerup which enables it.

Game's primary obstacles are gaps and various enemies, from which the most common are *Goombas*, *Koopas* or *Flowers* (Figure 1.2). All enemies defeat Mario if they touch him. Most of the enemies can be defeated by jumping on them, but some will defeat Mario even if jumped on (e.g. Spiky and Flower).



Figure 1.2: Subset of enemies appearing in Super Mario Bros game. From left to right: Goomba, Red Koopa, Green winged Koopa, Spiky, Bullet Bill Blaster, Flower in a pipe and Bowser.

Another obstacle in the game are pipes, which need to be jumped over, but can also contain secret passages. Some pipes contain Flowers which are appearing and disappearing in intervals. The game also contains platforms made from *bricks*, which can be destroyed if Mario hits them from the bottom, or *question mark blocks*, which can contain *coins* or different *power ups*. Coins can appear also outside of these blocks, and if the player acquires 100 coins he gets one more life. List of powerups appearing in Super Mario Bros game and their descriptions can be seen in Table 1.1.

To finish a level, Mario must reach its right-most end without losing all his remaining lives. The levels are split into eight worlds, each containing four of them. After finishing fourth level of the eighth world by defeating the game's main antagonist, *Bowser*, the game is successfully finished, and Mario happily reunites with the princess.





Powerup type	Bonus	Image
Super Mushroom	Makes Mario one tile higher, and adds him one hit point.	
Fire Flower	Enables shooting and adds one hit point to Mario. They will appear instead of <i>Super Mushrooms</i> if Mario has already picked up one and has not lost its effect by losing hit points.	
Super Star	Makes Mario invincible for a short time.	
1-Up Mushroom	Adds one life to the player.	

Table 1.1: The table of powerups, that can appear in Super Mario Bros game, with their descriptions.



Figure 1.3: The title screen of the original Super Mario Bros game.

1.1.3 Mario AI Framework

Mario AI Framework [7] is an open-source implementation of Super Mario game. It was implemented as a part of tournaments of Super Mario AI players and PCG generators of levels. The game simulator is based on *Infinite Super Mario* by Markus Persson, which is only slightly different than the original Super Mario Bros game. It uses different tilesets and some game rules are taken from other Super Mario games. The most notable changes are Flowers going up more than one tile, Koopas can be picked up by Mario and some missing powerups (1 UP and invincibility). It also does not support all of the Super Mario Bros enemies. The supported ones are: *Goomba*, *green Koopa*, *red Koopa*, *Spiky*, *Flower* and *Bullet Bill Blaster*, which include also *winged* versions of Koopas and *flying* version of Goomba.

During implementation of this thesis, a newer version of the platform came out called *10th Anniversary Edition*. However, the implementation was in its later stages, so we stucked with the older one.

1.2 Procedural Content Generation in games

Procedural Content Generation is the algorithmical creation of game content with limited or indirect user input [8, Chapter 1]. By this definition, PCG can be combined with hand crafted material. This approach is called *mixed initiative* [8, Chapters 1, 11]. However, in our work we will be generating content solely algorithmically.

The most common usage of PCG today is in video games. It is used to generate a wide variety of content, using many different techniques [3]. Using PCG, we can also generate levels for platformer games [9] [10]. This will allow us to have a practically infinite number of levels in the game.

There are various approaches for generating levels. Some of the following terms are not well standardized yet, so we will be using terminology from *PCG lecture at Charles University* [11]:

- **Building blocks** - we create a library of various blocks of levels (e.g. chunks) and combine them to create levels [12],
- **Template instantiation** - similar to building blocks, but the blocks are only templates, which, when instantiated (put into level), are varied in some ways,
- **Multipass** - the level generation is done in multiple phases. In each of them, the algorithm passes through the level and generates different features. For example, one phase can generate terrain, enemies or items [13],
- **Grammars** - we apply the grammatical evolution algorithm [14, 15] on a custom grammar whose symbols represent parts of a level. Result of the evolution is a sentence of the grammar, from which a level is constructed.

In our work, we will be generating levels using *building blocks* and *multipass* approaches.

1.3 Artificial Intelligence in games

Artificial Intelligence is a set of methods, which try to think or act humanly or rationally [16, Chapter 1]. While it is used in many computer science fields, in this thesis we will be concerned specifically about AI in games.

One concrete use of AI in games is to control game entities or even non human players [17, Chapter 3]. Some games use this kind of AI to let human players play against (or with) another players without the need of additional human players. This is very useful for e.g., *sport* games, *shooter* games or *real-time strategies*. These players can also be used to evaluate PCG content [17, Chapter 4]. They are useful for this purpose because we can use their performance on the generated content as an evaluation metric. AI-controlled players are more preferable than human players in our case since we will need thousands of evaluations on many different levels.

1.4 Artificial neural networks

Artificial neural networks (ANNs) are structures inspired by real-life human neural system. Just like their biological counterpart, they consist of a set of neurons and oriented connections between them. In the following sections we will describe them as they are defined in *Deep Learning Book* [18].

1.4.1 Neuron

Each connection in an ANN has its own *weight* and every neuron has its own computed value. To calculate it, we firstly compute a weighted sum of values of the neurons connected to the concerned node. Then, we apply an *activation function* to the sum and its output is the value of our neuron. Any (one dimensional) function can be used as an activation function, but there are only a few of them which are commonly used. The most widely used ones are the following:

- **ReLU (rectified linear unit)** - $ReLU(x) = \max(0, x)$
- **Sigmoid** - $\sigma(x) = \frac{1}{1+e^{-x}}$
- **Tanh** - $\tanh(x) = \frac{2}{1+e^{-2x}} - 1$
- **Identity** - $id(x) = x$

An example of computing an activation function can be seen in Figure 1.4. Neurons, whose values are above a given threshold, are commonly called *activated*.

The only neurons, whose values are not computed are *input neurons*. These are assigned an instance of a problem being solved by the network. *Output neurons* then contain output of the network when all neurons' values are computed. Usually, only activated output neurons determine the network's output.

There are multiple types of ANNs which are generally used. In the following sections we will describe those, that we will use in our work.

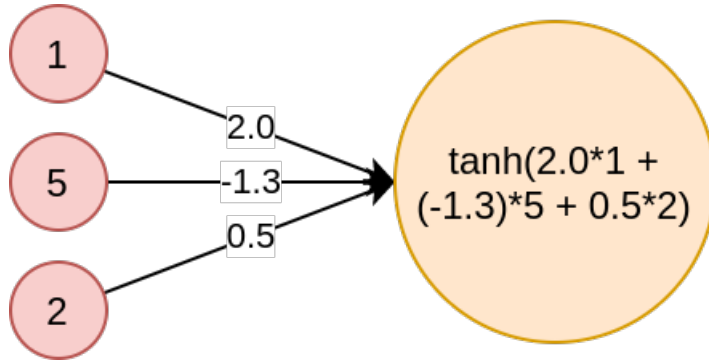


Figure 1.4: Computation of a neuron's (orange) value with \tanh activation function and three neurons (red) with already computed values connected to it. Firstly, we compute weighted sum of values of connected neurons (using weights from connections) and then apply activation function.

1.4.2 Feedforward layered networks

Feedforward layered networks [18, Chapter 6] are a special kind of ANNs with two distinctive features. They are:

- **layered** - all the neurons are split into disjoint sets called *layers*. Generally, we distinguish three types of layers. *Input layer*, which contains all the input nodes and no others, *output layer* containing all the output nodes and no others. The other layers are called *hidden*, primarily because they do not interact with the outer world,
- **feedforward** - values from the neurons are always fed forward to one of the later layers and never to itself or a previous layer.

There exist multiple types of connections of neurons between layers. In our work, we will be working only with the *dense* connection. This type of connection between two layers specifies that each neuron from one layer is connected to each neuron from the next layer. A simple neural network with dense layers can be seen in Figure 1.5.

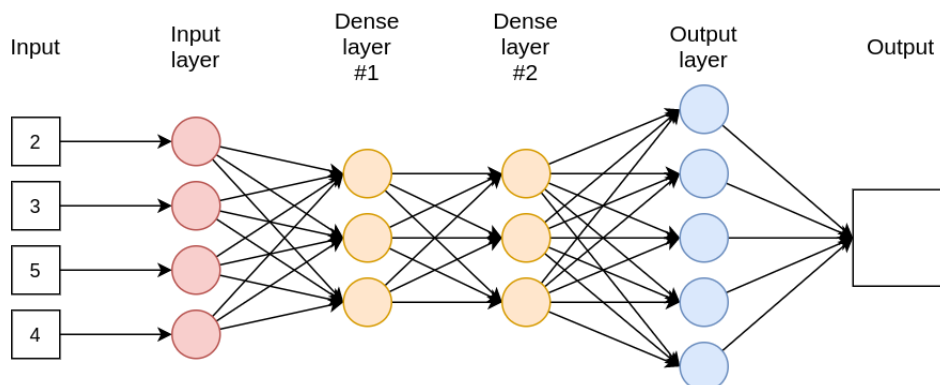


Figure 1.5: An example of an artificial neural network with input layer (red), two hidden dense layers (orange) and an output layer (blue).

Feedforward layered networks are used to approximate a high dimensional function. They usually perform very well on classifying tasks [19].

1.4.3 Recurrent networks

Recurrent networks are used for processing sequential data, where some kind of state needs to be preserved between processing of the individual sequence elements [18, Chapter 10].

These networks got their name because of the fact that they contain recurrent connections between nodes. We compute values of these nodes a little differently than in *feedforward* networks. The computation involves not only value received from the previous node(s), but also values the neuron computed in previous steps. After calculating current value, an activation function is applied to it, just like in feedforward networks, which results in the node's value for the current element in the sequence.

As already mentioned, this architecture is designed for processing sequential data where there are relationships between consecutive elements. They perform well, for example, on *natural language processing* tasks [20].

1.4.4 Artificial neural networks in games

ANNs can be used to control AI players in games. Recently, they are getting pretty successful, especially deep neural networks (networks with many hidden layers or nodes). They were successfully trained to superhuman performance in some classical board games [21]. In recent years, deep reinforcement learning is getting to the state-of-the-art level in even more of these games [22]. This technique was also used to train reactive agents [23]. However, in this work we will not be using deep ANNs. Our networks will not have more than 2 hidden layers, or more than a few tens of neurons (excluding input and output neurons).

1.5 Evolutionary algorithms

Evolutionary algorithms (EAs) are optimization algorithms based on real-world biochemical processes and Charles Darwin's evolution theory [24, Chapter 3]. As optimization algorithms, they try to find the best possible solution for a given problem.

They begin with a number of randomly generated solutions, also called *individuals*. Each one of them is then evaluated using a user-defined *fitness function*, which computes how well the individual is performing in solving the given problem. Then, tuples (pairs, triplets or larger) of individuals (*parents*) are selected for recombination which creates a new individual (*offspring*). These are then modified using random *mutations* and evaluated. Finally, some of them are selected to *survive* to the next cycle and the process repeats from the selection parents again.

This cycles, also called *generations*, end after a given termination condition. One of the most common conditions is a fixed number of generations. After that, the individual with the highest value of fitness function is selected as the best found solution to the problem. This general scheme of evolutionary algorithms is depicted in Figure 1.6.

There exists a large number of types of evolutionary algorithms. They differ in representation of individuals, fitness computation, can contain additional steps or

compute mutation and crossover differently. In our work, we will be using mostly *genetic algorithms* or genetic algorithms with a small adjustments.

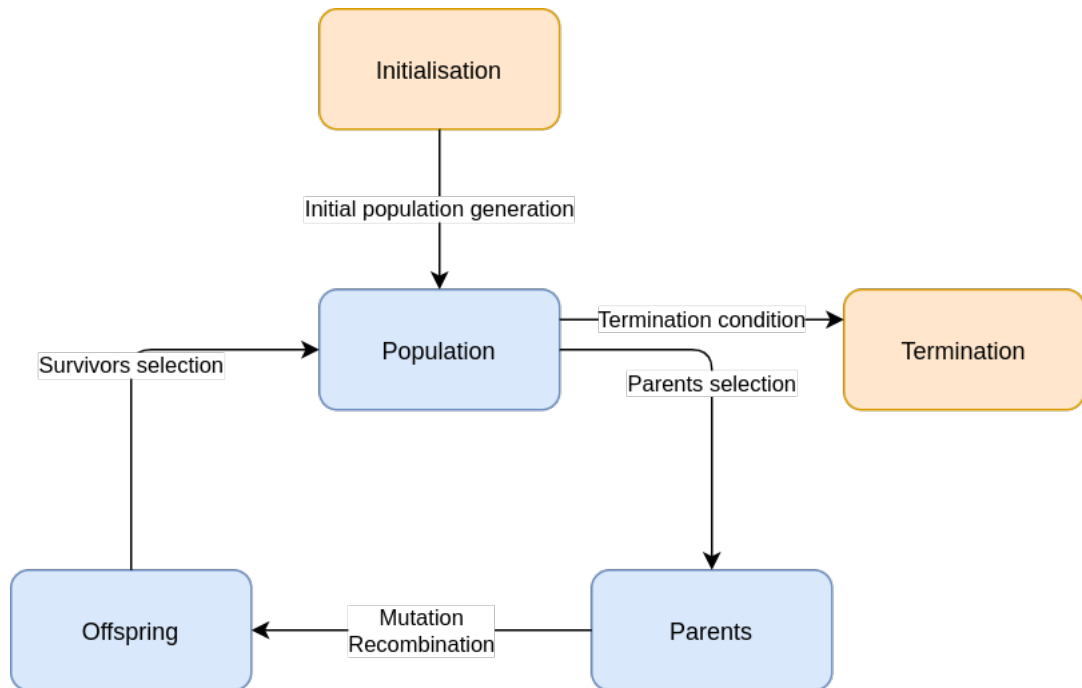


Figure 1.6: General scheme of evolutionary algorithms.

1.5.1 Genetic algorithms

Genetic algorithms (GAs) are a special kind of evolutionary algorithms. They have multiple variations, but the simplest form is called *Simple Genetic Algorithm* (SGA), conceived by John H. Holland [24, Chapter 6]. Individuals in SGA are *bit-strings*, mutations randomly flip a gene’s value and they use *1-point crossover* for recombination. This operation takes 2 random individuals, splits them on a random index and swaps one part of the genes between the parents. *Roulette wheel selection* is used for selecting parents. It assigns each individual a probability relative to its fitness value and randomly picks one.

SGAs, however, have some flaws. It is not always appropriate to encode our solutions as bit-strings, so some GAs use non-binary representations. Generally, GA individuals are 1D arrays of any type.

There are also various mutations we can perform in general GA, some of which are:

- **standard mutator** swaps a gene’s value with probability p to a random new value,
- **gaussian mutator** selects a random value from normal distribution (with given mean and standard deviation) and adds it to the gene’s current value with probability p .

GAs also use multiple types of crossovers, some of which are:

- **n-point crossover** splits individuals on n places and swaps all even or odd parts between them (an example of *2-point* can be seen in Figure 1.7),

- **uniform crossover** chooses each gene uniformly randomly from one of the parents.

Another downside of SGA is its survivor’s selector, where only offspring are selected for the next generation. In generic GA, we can use *elitism*, which is a selector preserving the best individuals from the parents population. Roulette wheel selection can also be replaced by different parents selectors:

- **Stochastic universal sampling (SUS)** can be imagined as splitting $[0, 1]$ interval into smaller intervals representing individuals. Each individual’s interval size is proportionate to its fitness. Then, we choose individuals, whose intervals contain points k/n , where n is the required number of selected individuals and k acquires integer values from 1 to n . This selection is preferred to Roulette Wheel selection because it yields better sample of the parents distribution [24, Chapter 5],
- **Tournament selection** of size k selects k individuals, plays a tournament between them and selects the winner as one of the parents. This is repeated until we have enough parents. Sometimes, we specify probability p for selecting the winner as a parent and lesser probability for selecting one of the others. This selection can have multiple advantages over SUS and Roulette Wheel selections based on the scenario in which it is used:
 - adding (multiplying by) a constant to the fitness function does not affect the selection’s results,
 - the fitness does not need to be quantifiable, we just need to be able to compare 2 (or more) individuals,
 - we do not necessarily need to compute fitness of all the individuals.

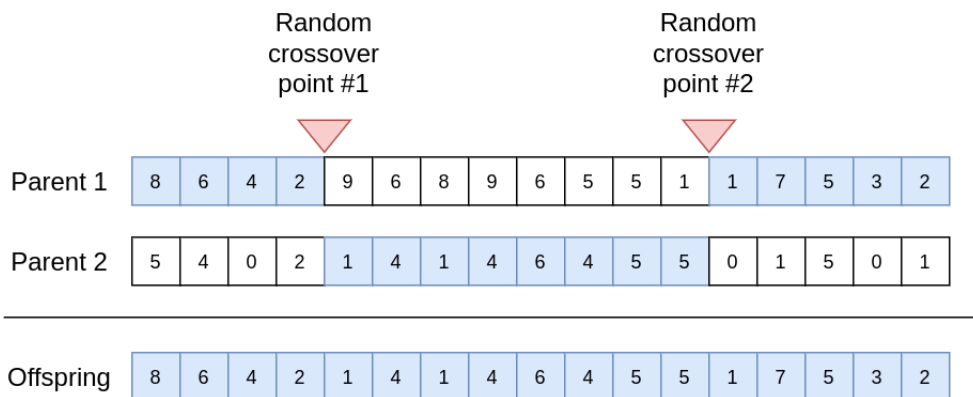


Figure 1.7: An illustration of 2-point crossover. In this example, the indexes where the individuals are split are randomly chosen to be 4 and 12.

1.5.2 Neuroevolution

Neuroevolution is a class of evolutionary algorithms, where individuals represent ANNs. It was used in a few games for various purposes [25]. For example, in *The*

Open Racing Car Simulator it is used to control player [26], while in *Galactic Arms Race* it is used to generate game content [27].

There are multiple types of neuroevolution algorithms. They differ in what kind of ANNs they are evolving and which parts of them are they evolving. In our work, we will be using a simple algorithm which we call *neuroevolution of weights* and also *NEAT* algorithm [28].

Neuroevolution of weights

Probably the most straightforward implementation of a neuroevolution algorithm is to evolve only weights of a given network with non changing topology. We will take all the weights of the ANN and put them in a one dimensional array. Such arrays with different values will represent networks with the same structure but different weights. They can be then used as individuals in a GA.

NEAT

Neuroevolution of Augmenting Topologies (NEAT) [28] is a neuroevolution class algorithm which evolves weights as well as structure of ANNs.

The individuals in this algorithm are lists of connections, where each gene represents one connection. A connection defines the in-node, the out-node, the connection's weight, whether the connection is enabled or disabled and its innovation number (Figure 1.8). The initial population consists of minimal neural networks with only input neurons which are directly connected to output neurons.

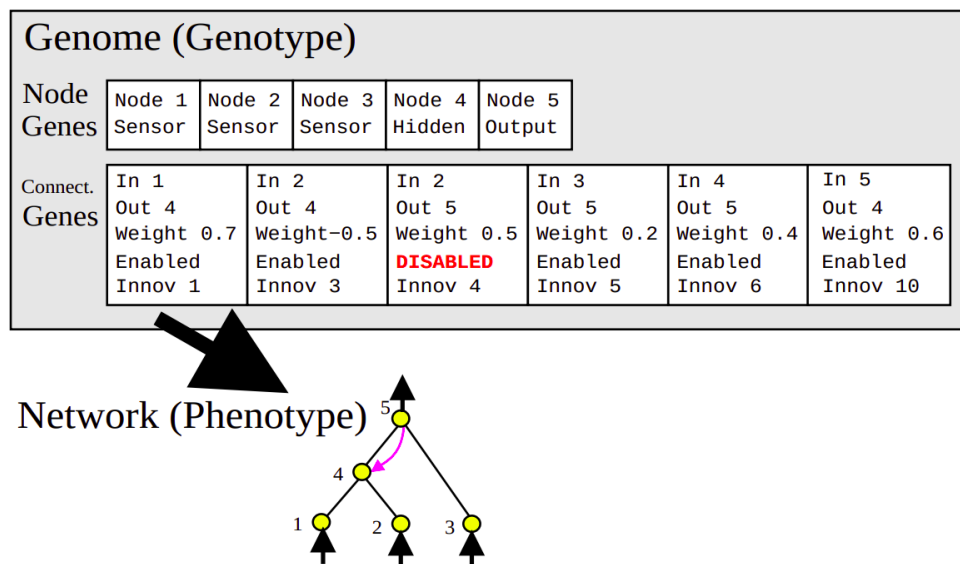


Figure 1.8: An example of NEAT individual (top) and the ANN it represents (bottom). The genome in the implementation contains only the list of connections, the list of nodes is redundant. The image is taken from the original paper about NEAT [28].

The algorithm uses three *mutation* operators:

- **weight change** - randomly perturbs weight of a connection,

- **add a connection** - chooses two random nodes which are not yet connected and creates connection (new gene) between them in the individual. This mutation can also create a *recurrent connection* by connecting a node to some other node which appears topologically before the first selected node,
- **add a node** - adds a new node to the individual. This is done by randomly choosing an existing connection, disabling it, and then creating two new connections: one from the original in-node to the newly created node and the second one from the new node to the original out-node.

Crossover in neuroevolution is not a simple operation. Two different networks solving the same problem may have very different topology, and solve the problem differently. Because of this, if we just choose random connections from two parents and create a new offspring from them it would result in a useless network in most cases.

To solve this issue, NEAT keeps track of every gene's historical origin. The assumption here is that the genes with the same origin will be solving the same subproblem in the network. The tracking is done by storing a *global innovation number* and whenever a new gene is created, NEAT assigns it the current value of that number and increases it. The innovation numbers of genes therefore represent chronology of genes' appearances.

With each connection (gene) having assigned an innovation number, NEAT can perform crossover between two individuals. Firstly, it divides the genes into three groups:

- **matching genes** - those, whose innovation numbers appear in both parents,
- **disjoint genes** - those, whose innovation numbers appear only in one of the parents, and are *inside* the range of the other parent's innovation numbers,
- **excess genes** - those, whose innovation numbers appear only in one of the parents, and are *outside* the range of the other parent's innovation numbers.

When creating a new offspring, we randomly take one gene from each pair of *matching* genes. *Disjoint* and *excess* genes are taken from parent with higher fitness. In case of parents having the same fitness, they are taken randomly. Visualisation of the gene division and the crossover process can be seen in Figure 1.9.

1.5.3 Coevolution

Coevolution has its roots in nature again. We will use its definition from *Coevolutionary Computation* article by Jan Paredis [29]. Coevolution differs from previously described evolution in that it evolves more than one population at a time and they constantly interact with each other throughout the evolution. It can be divided into two types:

- **cooperative** - fitness of one population increases when fitness of some other population increases. Thus, it is beneficial for the populations if any of them increase their performance,

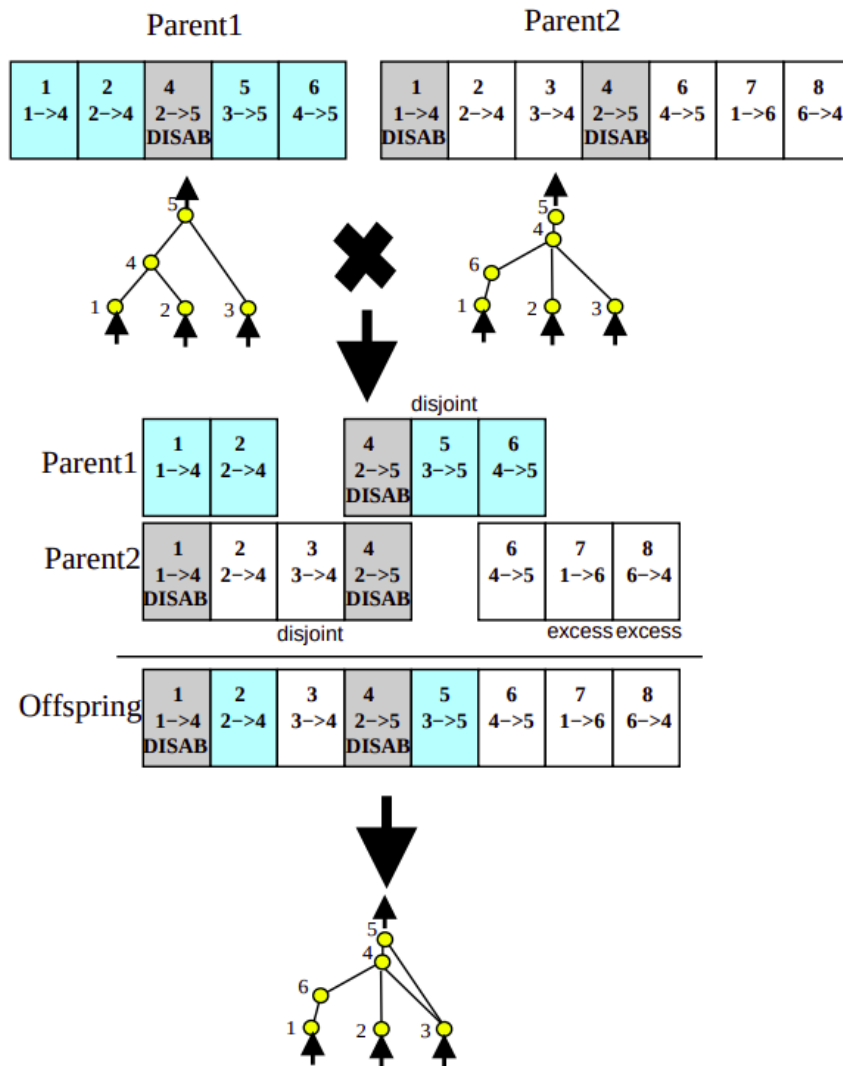


Figure 1.9: An illustration of crossover in NEAT algorithm. On the top we can see the two parent networks. The middle part displays alignment of connections with the same innovation numbers for better illustration. The bottom part displays the resulting offspring network. The parents are assumed to have the same fitness, so excess and disjoint genes from parents are chosen randomly. The image is taken from the original paper about NEAT [28].

- **competitive** - the populations are competing against each other. Increase of a fitness in one population leads to decrease of fitness in another population. However, the goal is that by repeating this process, the overall objective of the whole environment will be improving.

If we observe nature we can find many examples of both of these types. Typical example of competitive coevolution would be populations of *predators* and *preys*. While *preys* evolve their survivability techniques (camouflage, running speed, etc.), the *predators* must adapt to these changes and they evolve better hunting abilities (longer claws, better eye-sight, etc.). An example of cooperative coevolution can be seen between many kinds of plants and bees.

Simulating this nature behavior in computer science has already proven superior to single evolution for some problems [29] so it makes sense to try it for other problems. In our work, we will be using competitive type of coevolution.

1.6 Markov chains

In this section we will define Markov chains, which we will use later for levels generation.

A *stochastic process* is a collection of random variables:

$$\{X(t), t \in T\}$$

where t is usually understood as time, and $X(t)$ as a state of some environment at time t [30, Chapter 2.9]. A Markov chain is such stochastic process, where the value of the state in the following time step is dependent only on the value of the state in the current time step. More formally, let $\{X_n, n = 1, 2, 3, \dots\}$ be a stochastic process, where X_i are independent random variables and each acquires finite or countable number of values. We then say, that the process is a Markov chain if the following equation holds

$$P(X_{n+1} = j | X_n = k) = P(X_{n+1} = j | X_n = k, X_{n-1} = i_{n-1}, \dots, X_0 = i_0)$$

for each j, k , sequence of states i and $n \in \mathbb{N}$ [30, Chapter 4.1]. We can then define $P_{kj} := P(X_{n+1} = j | X_n = k)$ and put these values to a (possibly infinite) matrix:

$$\mathbf{P} = \begin{pmatrix} P_{00} & P_{01} & \cdots \\ P_{10} & P_{11} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

Matrix \mathbf{P} is called *transition (probability) matrix*. Each row i of such a matrix then specifies probabilities of a transition from state i to every other state. This means, that the sum of probabilities in each row must be 1, because the probability of transitioning from one state to any state is 1.

When we will be using this transition matrix in our algorithm to generate a sequence of states, we will firstly need to generate the initial state. For this purpose, it is desired to define a distribution of the initial state [30, Chapter 4.2]:

$$\alpha_i := P(X_0 = i), i \geq 0, \sum_{i=0}^{\infty} \alpha_i = 1$$

We will call a vector of value α_i an *initial state (probabilities) vector*.

2. Analysis

In this chapter, we will analyze our options to achieve the goals we stated in Introduction chapter. After that, we choose approaches which we will use and also mention some high level decisions for implementation, such as chosen programming language and used 3rd party libraries. More detailed description of our implementation is presented in Chapter 6.

2.1 AI player evolution

In order to implement our coevolution, we must decide which evolutionary algorithms to use for AI players and level generators in the first place. We will start with AI players evolution. Firstly, we need to choose, what type of an agent we want. We can select from two kinds of agents:

- **reactive agent** decides which action to play at each game step based only on the actual state of the game. Therefore, its decision making process is fast and the agent can quickly react to changes in the environment,
- **lookahead agent** chooses which action to play using actual game state and a number of simulated future states. Because this process is more difficult then the previous one it takes more time, but the agent's performance is potentially better.

We have decided to choose reactive type agent, because Super Mario game's environment is dynamic and requires fast reactions to its changes. Levels of the original game are also designed in a way that they do not require planning ahead to be solved (e.g. they do not contain dead ends and overcoming enemies is pretty straightforward).

Reactive agents can be implemented using various techniques. Some of the common ones are the following:

- **rule-based** - as the name suggest, these agents are given a set of *if-then* rules according to which they will behave (e.g., if the next tile is a gap, jump),
- **finite state machine (FSM)** [17, Chapter 2] - there is a defined set of states, in which the agent can be, and a list of transitions which define conditions to move between states. In each state, the agent has defined actions to perform,
- **behavior trees** [17, Chapter 2] - the agent is controlled by a special kind of oriented tree graph. In this tree, each leaf represents an action the agent can perform. In each game tick we traverse through the tree (starting from the root node) and let the agent perform the action from the leaf node where we ended. When traversing through the tree, every parent node selects (according to given rules) to which child to continue,
- **ANN based** - an artificial network (Section 1.4) is evaluated at each game step, and its output chooses which actions the agent will play.

Because ANNs are most easily trained from the mentioned methods and scored pretty good results recently in AI in games, we chose to use this technique. It is also a pretty general and can be easily reused in other games, while the others would need to be reimplemented for each game.

When using an ANN-controlled agent, we can use neuroevolution, reinforcement learning (RL) or some kind of supervised learning. Supervised learning in our context is practically impossible, because our agents will be trained on generated levels. We chose neuroevolution over RL, since it is more natural to use EAs in coevolution than *gradient descent* techniques and because of previous familiarity.

It is difficult to see which algorithm should provide the best results beforehand, so we chose to implement two of them, and then compare their performance and suitability for coevolution. The first one is neuroevolution of weights and the second one is NEAT (Section 1.5.2).

2.2 Level generator evolution

Like with AI players, there exist multiple approaches for generating levels in 2D space. Some of these are *building blocks* approach, *grammars* or *multipass* approach (Section 1.2). Again, it is not easy to see, which algorithm should work better, so we decided to implement two algorithms, compare them and choose the one which is performing better.

We decided to implement one algorithm based on building blocks principle. It is one of the most basic approaches and already had nice results [12]. Using this approach we will create a library of chunks, which we will then use to generate levels. To combine these chunks we will use a Markov chain and an EA to evolve its transition matrix and initial state vector.

The second approach we chose is multipass generation. It is a pretty common approach for levels generation for platformers and also won one of the Mario AI tournaments [13]. In this algorithm, we will generate each kind of obstacle that can appear in the game in a different stage while traversing the level from left to right. Each obstacle type will have its own probability to appear at the current location. Since Super Mario game does not contain many obstacles, we will not have to implement many passes.

Because both chosen algorithms can be configured using different set of probabilities, we will use evolutionary algorithms to find the most suitable ones.

2.3 Coevolution

Our goals are to evolve well performing AI player and specific level generators for Super Mario game. We propose to achieve this by using coevolution of these using algorithms selected in sections 2.1 and 2.2.

The evolutions of AI players and level generators will be taking turns for some number of generations. When evolving AI players, we will try to evolve such, which solve as many levels generated by the currently best level generator as possible. When evolving level generators, we will try to evolve such, that generate levels which are adequately difficult for the currently best AI player.

This way, when we will be doing the AI players evolution, it may be easier to learn on adequately difficult levels than on very difficult ones for the current players. When switching back to the level generators evolution, we should have better AI players, so it should result in level generator which generates more difficult levels than the previous one.

After each level generators evolution run, we will store the best one. When the whole process finishes, we should have a list of generators, which generate gradually more and more difficult levels. The last evolved AI player should also be able to solve these.

From the definition of our coevolution, we can see that this will be a *competitive* type coevolution (Section 1.5.3).

2.4 Game engine

In this work, we will use Super Mario engine from *Mario AI Framework* [7]. This engine was implemented to be used for benchmarking AI agents playing the game and generators generating levels for the game. It was already used in multiple AI and PCG tournaments so it should be capable of handling our needs.

We will use Jakub Gemrot's more polished version of the framework [31], because it is easier to work with¹.

2.5 Kotlin

Since we decided to use pre-existing game engine, which is written in Java language, we are bound to use a JVM based language for our implementation. The most straightforward option is to use Java. However, even though Java is getting more attention from its developers in recent years and getting modernized, we still find it a little outdated language which lacks some of the modern concepts.

There are multiple alternatives for Java that can be used like *Scala*, *Kotlin* or *Groovy*. Kotlin is a modern language running on JVM² which was first released in 2011. It has been developed by *JetBrains* and is 100% interoperable with Java.

We chose *Kotlin* because it uses many modern programming constructs (null safety, smart casts, functional programming and more) which Java does not (or only partially) support. Its syntax is also more compact contributing to faster development and more readable code.

2.6 Third party libraries

Some of the algorithms and structures we will use in our implementation were already implemented as libraries by other developers. List of all 3rd party libraries we used (including their version) can be seen in Table 2.1.

We decided to use *Jenetics* library for evolutionary algorithms and *Deeplearning4j* for neural networks. Since these libraries have been around for a longer time

¹The newer *10th Anniversary* version, which was released after we started working on this thesis, is probably also quite polished.

²Currently, Kotlin can be compiled also to JavaScript (Kotlin/JS) and to native binary (Kotlin/Native), which supports multiple platforms.

and used by many users, their implementation should have minimum number of bugs. For NEAT algorithm we decided to use *evo-NEAT* [32] library because of its relative ease of use and available source code.

For displaying and storing different charts we decided to use *XChart* library, which is capable of displaying line charts and their realtime updating, which can be useful during evolutions.

Name	Version	URL
Mario AI Framework	0.3.1, Jakrub Gemrot's version	https://github.com/keflk/MarioAI (accessed 7.7.2020)
evo-NEAT	master@ef9f2d0	https://github.com/vishnugh/evo-NEAT (accessed 7.7.2020)
DeepLearning4j	0.9.1	https://deeplearning4j.org/ (accessed 7.7.2020)
nd4j	0.9.1	https://deeplearning4j.konduit.ai/nd4j/overview (accessed 7.7.2020)
DataVec	0.9.1	https://deeplearning4j.konduit.ai/datavec/overview (accessed 7.7.2020)
sl4j	1.7.28	http://www.slf4j.org/ (accessed 7.7.2020)
jenetics	6.0.0	https://jenetics.io/ (accessed 7.7.2020)
XChart	3.6.3	https://knowm.org/open-source/xchart/ (accessed 7.7.2020)
HuffmanCoding	master@b9613f7	https://github.com/marvinjason/HuffmanCoding (accessed 7.7.2020)
JUnit	5.6.2	https://junit.org/junit5/ (accessed 7.7.2020)
Hamcrest	2.2	http://hamcrest.org/ (accessed 7.7.2020)
MockK	1.10.0	https://mockk.io/ (accessed 7.7.2020)

Table 2.1: The list of all 3rd party libraries we used in our project.

3. AI player evolution

We start describing our evolutionary algorithms in more detail with evolution of AI player. Firstly, we will mention some related works, and then we will look at our two approaches - the neuroevolution of weights and NEAT. Then, we will follow with describing our results with these players and finally, we will draw conclusions from them.

3.1 Related works

AI in games is a very large field and so we will mention only works which concern directly Super Mario game. Some more generic work in field of AI players for games was already mentioned in sections 1.4.4 and 1.5.2.

Many works were submitted for *Mario AI tournaments* [6], where multiple different approaches were used. The best scoring approaches were state search via A^* algorithm (won the 2009 tournament) and another search via A^* , which used actions evolved beforehand using EAs (won the 2010 tournament). EA and ANN approaches performed poorly in comparison with the others, according to the paper. However, coevolution may yield better results and our aim is not for perfect AI but multiple AIs with increasingly better performances.

In other work, the authors compared neuroevolution based agents, where one used feedforward layered network, the second a recurrent ANN, and the third *HyperGP* algorithm [33]. The result of this work was that the agents were able to solve levels with “occasional gaps and healthy number of enemies”, but had “problems with generalization”. All three agents achieved similar results, while agents with smaller ANNs (less layers and neurons) performed better than agents with larger ANNs. This counter-intuitive result may be caused by higher dimensionality of search space, which makes the ANN more difficult to train, or overfitting.

There were also attempts to evolve behavior trees for a Super Mario agent [34]. According to the results shown in the paper, the resulting evolved agent performed similarly to the other EA based agents in a competition mentioned in the work.

3.2 Our approach

In Chapter 2, we decided to implement two reactive agents based on ANNs. To use this approach, we must first design the networks we will use:

- **Input** - in order for an ANN to be able to decide which action to play, it must know how the environment around Mario looks like. Therefore, our input will contain one 2D grid of tiles surrounding Mario and second one containing enemies. The grids will contain immediate surroundings of Mario.

However, since the tiles and enemies, which are behind Mario pose almost no threat and Mario does not need to know about them in order to progress in the level, we will offset the grid a little to the right (Mario will be on its

left end). This is illustrated in Figure 3.1. Thanks to this, Mario can see a few more tiles further in front of him while the grid size remains unchanged. Our input layer will contain neurons each representing one tile of one of the two grids.

Therefore, it will have $5 \times 5 \times 2 = 50$ neurons when using neuroevolution of weights and $7 \times 7 \times 2 = 98$ neurons when using NEAT (these exact values were chosen by results of the following experiments). They will acquire values based on the type of the tile/enemy which is on the given position. The specific values are taken from the Mario AI Framework (e.g. 0 - empty tile, -60 - impassable block, 80 - Goomba, etc.),

- **Output** - the output of our networks should decide, which actions the agent will play in a given situation. Therefore, the output layer will contain one node for each possible action the agent can play, which are *go right*, *go left*, *jump* and *shoot*, resulting in 4 neurons. We excluded *sprint*, because it is not required to solve the levels and would therefore unnecessarily increase dimensionality of the search space,
- **Hidden nodes** - in case of NEAT algorithm, we do not need to decide how the hidden nodes will look like since they are being evolved inside the algorithm. For the neuroevolution of weights algorithm, we will use smaller networks, which had better success in the mentioned related works. They will contain one hidden layer with 7 nodes. We will use *dense* connection between each layer.

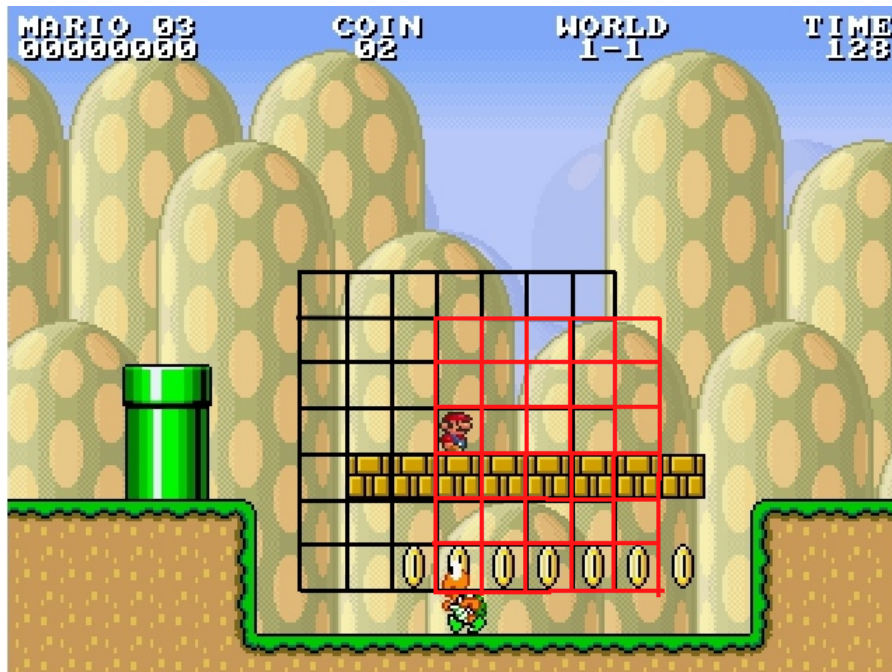


Figure 3.1: An illustration of which tiles (red) surrounding Mario the agent will see for grid size 5×5 .

3.2.1 Fitness function

The ultimate goal of our AI players is to solve as many levels our generators will generate as possible. Thus, we use the number of solved levels as an *objective function* for the agents. However, this function’s range is too sparse, and does not reflect that some agents may have solved larger portions of levels than others. Because of this, our fitness function will sum the distances the agent reached in the levels on which it was being evaluated.

After some experiments, we found out that the evolution tends to quickly find agents, which are going right and jumping all the time. This is because these agents will get pretty far in the levels and this strategy is pretty simple, therefore, the evolution will evolve them easily. However, they will also easily jump right into a gap or an enemy, and it seems to be difficult for the evolution to find agents which jump only when needed, so it gets stuck in this local maximum of the fitness function. To counter this issue, we implemented a second fitness function, which slightly penalizes the agents for each jump.

3.2.2 Training data set

Since we do not have our level generators yet, we will need to create some levels, which will be used for evaluation of our agents. We decided that original levels of Super Mario Bros could be a good benchmark, so we reimplemented 4 of them in the Mario AI Framework¹.

We decided to reimplement only some of the original 32 levels, because using all of them would slow down the evolution and not bring more new features. Some of the features are not even implemented in the framework². We chose levels of subjectively various difficulties and containing all features present in the framework. The levels we decided to implement are listed in Table 3.1.

Level	Brief level description
Stage 1 Level 1 (S1L1)	Pretty simple level, containing only a few gaps, Goombas, Koopas and pipes without Flowers
Stage 2 Level 1 (S2L1)	Contains more enemies than S1L1 and also pipes with Flowers and flying Green Koopas
Stage 4 Level 1 (S4L1)	Contains less enemies than S1L1, but contains also pipes with Flowers and Spikies
Stage 5 Level 1 (S5L1)	Similar to S2L1 but contains also Bullet Bills

Table 3.1: The table containing levels we decided to reimplement in Mario AI Framework and their brief description.

For better evaluation purposes, we also split the levels into multiple parts. This way, if a level contains an obstacle the agent cannot solve right at the beginning, but otherwise could solve the level, it will get more appropriate score by the fitness and objective functions. After this splitting, we had 23 level parts.

¹10th Anniversary version of the framework, which was released after we started working on this thesis, contains these levels, but the older one we are using does not.

²Examples of missing features are *lifts*, *fire bars* or some kinds of *enemies*. In the 10th Anniversary version they are already present.

We also implemented two artificial levels. The first one is *path-only level*, which contains only flat ground and no obstacles. This level is implemented to help the evolution find agents which learn to go the right. The second one is *gaps level*, containing only gaps of various widths unevenly placed. This forces the AI to learn when to jump correctly and penalize players which jump all the time.

We then used these 25 levels when evaluating the agents during evolutions.

3.2.3 Neuroevolution of weights

The first neuroevolution algorithm we are going to implement is the *neuroevolution of weights* (Section 1.5.2). When using this approach, we encode a network as an fixed-size array of the network’s weights. We use *gaussian mutator* with mean 0, and standard deviation 1. We decided not to use crossover in this evolution because we assume that combining weights of two different networks hardly creates a better network.

3.2.4 NEAT

We did not do any changes to the NEAT algorithm, meaning that it works exactly as described in Section 1.5.2.

3.3 Our results

To find the most suitable values of various hyperparameters, we ran multiple experiments. We will describe them in the following sections. The results of all of them (evolved agents, evolution charts) are a part of the electronic attachment (see Appendix A.1).

3.3.1 Experiments: Neuroevolution of weights

Before the final experiments, we fixed all the hyperparameters to values that can be seen in Table 3.2 after a few initial experiments. With these initial settings, we chose one hyperparameter, and run multiple experiments with various values of the hyperparameter and chose the one with which the evolution produced the best performing agent. Then, we chose a different hyperparameter and did the same thing.

This way, we have sequentially experimented with *mutation probability*, *population size*, *input grids size*, *hidden layer size*, *fitness function*³ and finally *network weights range*. Each of these experiments were run 4 times and their results were averaged. In Figure 3.2 we can see the results of these experiments. The resulting best values of the hyperparameters we found are listed in Table 3.3⁴.

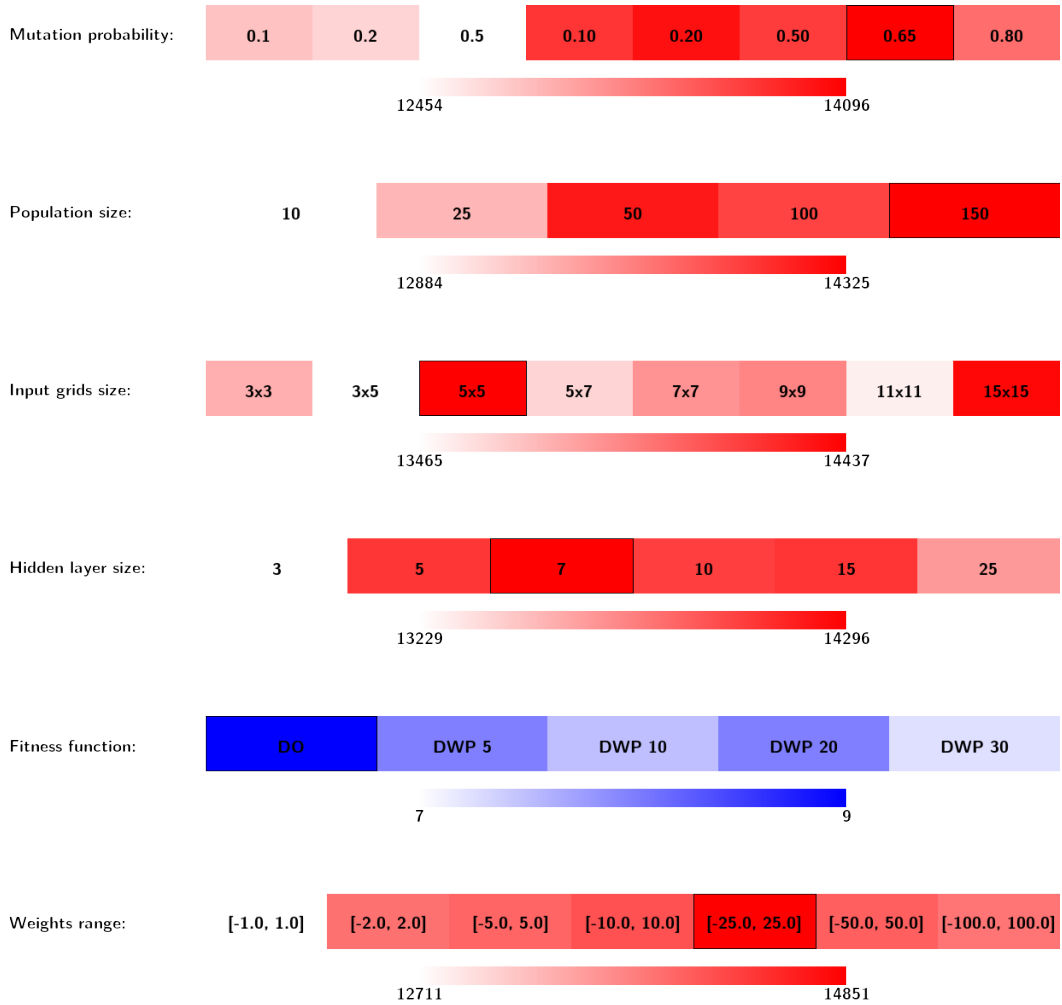
As for the reliability of these tests, it has a problem that a change in one hyperparameter may affect results of experiments with other ones. Ideally, in order to find the best hyperparameter values, we would need to run the experiments

³Comparing fitness values of different fitness functions is meaningless, so to choose the best hyperparameter value we used objective values.

⁴Even though when using population size 150 had a slightly better result, it was so insignificant that we stuck with population size 50 for which evolution runs much faster.

Neuroevolution of weights	
Hyperparameter	Value
Mutation probability	0.2
Population size	50
Input grids size	5x5
Hidden layer size	5
Fitness function	Distance only
Network weights range	$[-2.0, 2.0]$

Table 3.2: The initial state of the hyperparameters before experiments with the neuroevolution of weights.



Neuroevolution of weights	
Hyperparameter	Value
Mutation probability	0.65
Population size	50
Input grids size	5x5
Hidden layer size	7
Fitness function	Distance only
Network weights range	$[-25.0, 25.0]$

Table 3.3: The hyperparameters of the neuroevolution of weights with which we achieved the best results.

with each possible hyperparameter value combination. This method is called *grid search* and is sometimes used. However, it creates a *combinatorial explosion* in their number. Combined with the fact that using different hyperparameter values in our experiments did not change results too much, we decided to run them at least in this independent way.

3.3.2 Experiments: NEAT

Like with the neuroevolution of weights, we ran multiple experiments with NEAT algorithm too. Following the tradition, we were trying different values of hyperparameters one at a time. The initial configuration can be seen in Table 3.4 and the experiments were executed for *population size*, *input grids size* and *fitness function*⁵. The results of these experiments can be seen in Figure 3.3. The resulting best values of hyperparameters for NEAT we found are listed in Table 3.5⁶.

NEAT	
Hyperparameter	Value
Population size	100
Input grids size	5x5
Fitness function	Distance only

Table 3.4: The initial state of the hyperparameters before experiments with NEAT.

3.3.3 Additional improvements

After all these experiments, we tried two more improvements:

- **Dense input** - we made the input grid 2 times denser in each dimension, making one neuron represent quarter of the original tile. The idea behind this was that thanks to this approach, the agent would have a better sight of how close it is to enemies or gaps,

⁵To choose the best hyperparameter setting we compared objective values instead of fitness values.

⁶Even though when using population size 200 and 300 had a slightly better result, it was so insignificant that we stuck with population size 100 for which evolution runs much faster.

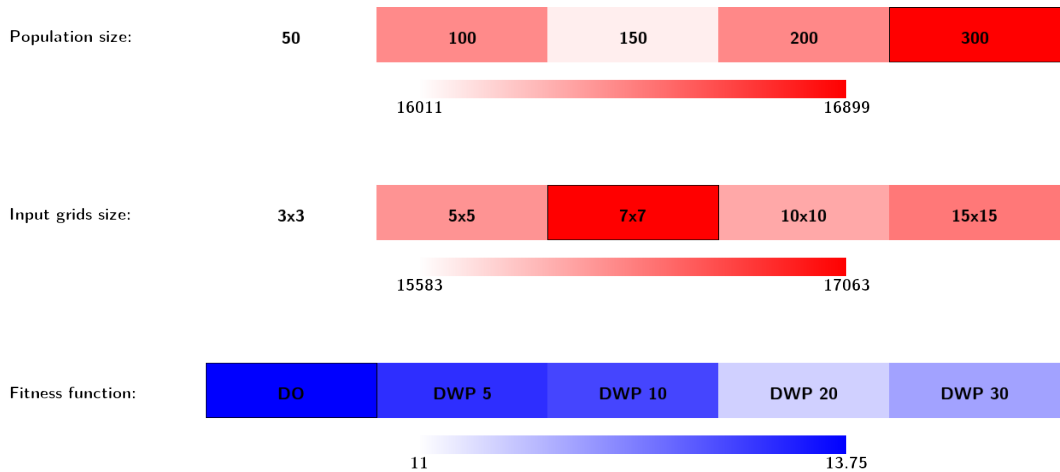


Figure 3.3: Results of running experiments with *NEAT* algorithm using various hyperparameter values. Each experiment (cell) was run 4 times and the results were averaged. Red color represents fitness values and blue one objective values (number of finished levels). The lighter the color is, the lesser value it scored, down to white, which is used for the lowest value. The highest value is always highlighted with a border. *DO* stands for *distance only* fitness and *DWP x* stands for *distance with penalties* fitness function using penalty x . We can see, that the best found values of the hyperparameters are: 100 (population size), 7x7 (input grids size) and *distance only* (fitness function).

NEAT	
Hyperparameter	Value
Population size	100
Input grids size	7x7
Fitness function	Distance only

Table 3.5: The hyperparameters of NEAT with which we achieved the best results.

- **One-hot encoded enemies** - we used *one-hot encoding* for enemies. Using this encoding, instead of having one tile represented by only one neuron in the input layer, it will be represented by as many neurons as we have types of enemies. These neurons will then acquire values either 1 or 0 based on whether an enemy of the given type appears in the corresponding tile. Using this technique, the AI player can better distinguish between enemy types appearing in the game.

However, after multiple experiments we came to the conclusion that these improvements had insignificant or no impact on the agent’s performance. Because of this fact, we decided not to use them in the future experiments to keep the network simpler to achieve faster evaluations and possibly also convergence in the evolutions.

3.3.4 Running times

All the experiments from this section were run on a machine with *Intel® Core™ i7-7700HQ* processor which has *4 cores* running at *2.80GHz* and uses *hyperthreading*, *16GB* memory and *Ubuntu 20.04 LTS* operating system. Both algorithms are implemented to evaluate individuals in parallel, so they can both exploit the advantage of having 8 threads.

The experiments, where the player was evolved by the neuroevolution of weights algorithm, usually converged in under 50 generations. Those, with the hyperparameter values we finally selected, ran in average a little more than *20 minutes*. When using population size 150 instead of 50, which had a slightly better results, the average time was almost *hour and a quarter*, so we decided to use population size 50 in the later coevolution instead.

The experiments with NEAT algorithm required much more generations to converge, but because the networks are smaller, they are evaluated faster. Those experiments with the hyperparameter value we finally selected took on average a little more than *one hour* to finish. We decided to use population size of 100, instead of 300, because those experiments ran more than 2.5 times longer.

Averaged evolution charts of the experiment using the best found hyperparameter values can be seen in Figure 3.4.

3.4 Conclusion

We have implemented an AI player for Super Mario game, which uses an ANN to decide what actions to play in each game step. Then, we used two neuroevolution algorithms to evolve the networks to get the best performing player. One of them was the *neuroevolution of weights* and the other one *NEAT*.

After that, we ran multiple experiments using various hyperparameter values to find out which of them work best. However, for the later coevolution, we have decided to use slightly different values than those, with which we evolved the best performing agents. The reason for this was, that the slightly better performance is too insignificant to pay for it with a much larger execution time.

Finally, players evolved by the neuroevolution of weights algorithm performed somewhat worse than those evolved by NEAT, but took much less time to con-

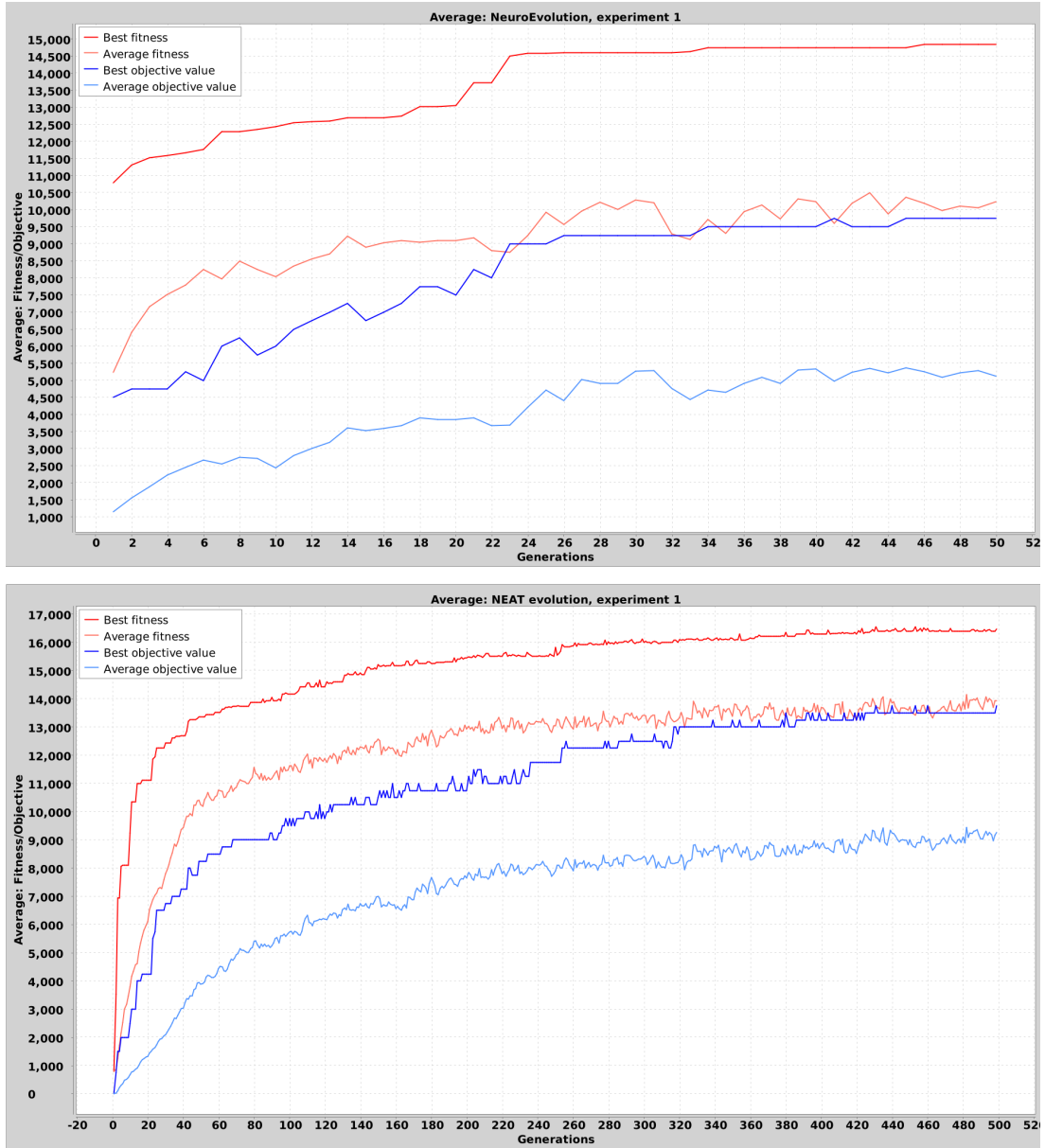


Figure 3.4: Averaged chart of 4 runs of the neuroevolution of weights (top) and NEAT (bottom) using the best found hyperparameter values. The fitness function is summed distance the player had reached on the evaluation levels and the objective function is the number of levels solved times 1000. We can see, that on average the best player solved almost 10 levels when using neuroevolution of weights and almost 14 levels when using NEAT. The chosen number of generation also seems to be enough for the evolutions to converge.

verge. This may make the first approach more suitable for coevolution, where the experiments will run even longer. However, because NEAT evolved players' performance is better, we choose to use both approaches in the final coevolution, but constrain the one using NEAT more (e.g., by running for less generations).

4. Level generators evolution

After implementing the evolution of AI players, we will implement one also for the level generators so we can combine them in one coevolution. Once again, we will implement two different algorithms to have better possibilities for the coevolution.

We will implement algorithms we specified in Chapter 2. Unlike ANNs for agents in the previous chapter, they can both be manually configured to generate feasible levels. However, the EAs will be used to fine-tune the generators for the player they are creating the levels for.

We will start by mentioning some previous works done in this field. After that, we will describe our level generators and their evolution in detail. Finally, we will showcase the results achieved by these methods.

4.1 Related works

Multiple works tried to generate levels for various games. Some tried level generation for general 2D games using the General Video Game AI framework (GVG-AI) [35]. In this work, the authors generated levels on a small 2D grid. They compared three approaches: *random* level generation, *constructive* level generation and *search-based* level generation. The random level generator generated the tiles randomly. The constructive level generator used the multipass approach and utilized game information retrieved from the framework. The search-based generator used the *feasible-infeasible genetic algorithm*. Finally, they let human players compare levels from each of the generators. In this survey, the *search-based* generator ranked best among these, as expected.

Others tried to generate platforms for platformer games based on rhythm [9] [10]. They combined smaller pieces of levels into larger ones aiming to make the player jump in a rhythm, which should help the player be in the *flow* [36]. *Launchpad* does this by firstly generating rhythms and then using design grammar to convert it into levels. The generator generates levels of various difficulties and obstacles. They are visually appealing and the generator has multiple parameters that can be configured from the outside.

One work tried to imitate the levels of Super Mario Bros game [37]. The authors created a set of 24 one tile wide columns that appear in the game (primarily in the first two levels) and used an EA to combine them to form a level. The individuals were sequences of these columns. Their fitness was computed based on how many and how difficult patterns from the original game they contain. These patterns were observed beforehand. To evaluate their results, they implemented another fitness function, which punished levels for containing patterns similar to those in the original game. Then, they let human players play through levels evolved by EAs using the different fitness functions. The survey found out, that the levels which were evolved to contain similar patterns as those in the original game are only slightly more fun and similar to the original game's levels than those evolved using the other fitness.

Finally, there were multiple works submitted to the Mario AI Championship's Level-generation track [6] [13]. Various techniques were used in these generators

including *building blocks* (could generate pretty complex levels but with occasional errors) and *multipass* generation (winner of one of the championships).

4.2 Our approach

Both our level generators are probabilistic. That means that they generate levels based on a list of given probabilities. It may be difficult to manually specify these probabilities so that the generators will generate levels with abstractly defined properties, like adequate difficulty or fun to play. For this purpose, we will use EAs. We will define fitness function that measures these properties and use an EA to find required generators.

The generated levels will contain all features commonly appearing in Super Mario games, which are primarily *gaps*, *platforms* and *pipes*. From the enemies, we decided to generate the following ones: normal *Goomba*, *green Koopa*, *red Koopa*, *green winged Koopa*, *Spiky* and *Bullet Bill Blaster* (Figure 1.2).

4.2.1 Chunked Markov Chain level generator

The first level generator we will implement is based on *building blocks principle* (Section 1.2). We will create multiple chunks, which will then be combined using Markov chains into one sequence of chunks. This sequence will then form a level.

Chunks

Firstly, we need to decide what our chunks will look like. If we look at some *Super Mario Bros* game levels, we will find out that they do not contain many different environmental features. There are primarily gaps, pipes, bullet bills, question mark blocks and brick blocks. And there is not much variation even in this small number of types of features. Gaps, pipes and bullet bills come with a few different sizes, because they are bound by how far (or high) Mario can jump. Additionally, platforms made of question mark blocks and brick blocks always appear at only two different levels above ground.

Thanks to this, we can make one chunk for each of the mentioned features which follow the stated constraints, and our library of chunks will still be pretty compact. It will contain the following chunks: *path* (of lengths 3, 4, 5, 6), *gap* (of lengths 2, 3, 4), *platform* (of lengths 1, 3, 5 using either bricks or question mark blocks), *pipe* (of heights 2, 3, 4), *bullet bill* (of heights 1, 2, 3, 4), *stairs* (of lengths 2, 3, 4) and *double platform* (of length 5, each platform being composed of bricks or question mark blocks). An example subset of these chunks can be seen in Figure 4.1.

The next thing we need to solve is how we are going to connect two neighbouring chunks. Since we already decided that they will be placed right next to each other, we only need to choose at which height they will be placed. We can't put the next chunk too high, because Mario would not be able to jump on it. When placing the next chunk lower than the previous one, theoretically, there is no limit on how much lower it can go so that Mario can still pass through these chunks. However, if we want to preserve the ability of Mario to go back in the

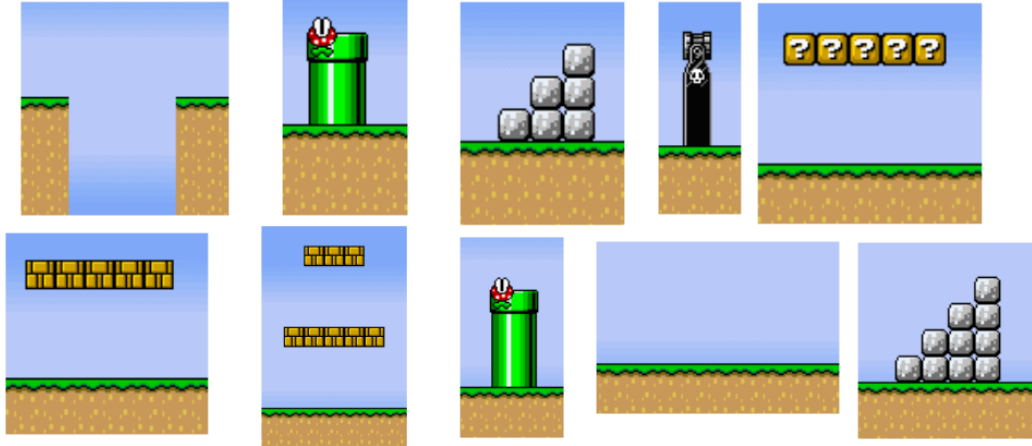


Figure 4.1: Example subset of chunks for our Chunked Markov Chain level generator. On the top row, from left to right, we can see the following chunks: gap of size 3, pipe of height 3, stairs of length 3, bullet bill of size 4 and single question mark platform of length 5. On the bottom row, from left to right, we can see these chunks: single bricks platform of length 5, double bricks platform of length 5, pipe of height 4, path of length 7 and stairs of length 4.

level we need to limit this change in height by the height which Mario can jump, too. Because Mario can jump 4 tiles up, the height change interval will be $[-4, 4]$.

Now that we have constrained the height change between two chunks, the generator still needs to decide its concrete values. We can let the generator randomly choose it between each two chunks. However, this way, we would not be able to configure the generator to generate e.g., only flat levels (which ones will be useful later in coevolution). So we will add a probability to the generator, which will tell how probable it is to change the height between two chunks, and then the actual height can be chosen randomly. Setting this probability to 0 will tell the generator to generate only flat levels.

The last thing to consider is how to deal with unsolvable levels. We could implement an AI player that can solve any solvable level (e.g., by using *depth-first search*). This has a disadvantage that such validation would be too slow. A second option is not to generate unsolvable levels at all. Thanks to our design of chunks, they can all be traversed by Mario. And thanks to our constraints to chunk placements, Mario will also be able to cross through each chunk switch. This way, our generator can not generate unsolvable levels.

Combining chunks using Markov chains

Now that we have defined how we are going to connect different chunks, we need to specify how the generator will choose among them when generating a new chunk. We propose to use a Markov chain (Section 1.6). It will contain one state for each chunk type. The initial state vector will specify probabilities that the given state (chunk) will appear first in the level. Then, element p_{ij} of the state transition matrix will define the probability to generate chunk j after previous chunk i .

Instead of Markov chains we could have one parameter for each chunk type which specifies its probability to appear in the level. However, Markov chains are more powerful in that they specify this probability for each pair of chunk types.

Now, we only need to know when to stop the level generation. This can be easily achieved by simply specifying the number of chunks the generator should generate. We will also hardwire the first and the last chunks to be empty paths for better game design [38].

Enemies

The next problem we need to solve is the generation of enemies. If we would place them directly into our chunks, the chunks would soon become too repetitive. To fight this issue, we could create different chunks with all the possible positions of all types of enemies. However, this way, the number of our chunks would grow considerably. Because of this, we decided to generate enemies independently of chunks generation.

After the level is generated, we will traverse through the level tile by tile, and on each tile, we will decide whether to generate an enemy or not. Each type of enemy will have its own probability of spawning. This way, we also increase the generator's configuration and possible levels space.

An example level generated by this generator can be seen in Figure 4.2. The probabilities in the Markov chain transition matrix and initial state vector were set to the same value. The probabilities of enemies were set to custom values.

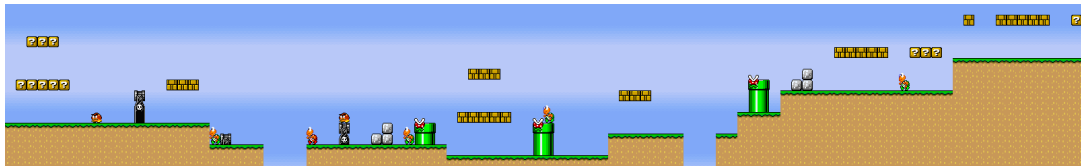


Figure 4.2: A part of a random level generated by Chunked Markov Chain level generator.

4.2.2 Multipass level generator

The second level generator we will implement is based on the multipass approach. We will fix its levels' length, and then the algorithm will traverse the currently generated level from left to right, tile by tile, multiple times. On each tile, the algorithm will generate some environmental features according to their defined probability. As discussed in Chapter 2, Super Mario Bros levels do not contain many different features, so each one can be generated in its own level pass. We propose the following passes:

1. **Ground pass** - firstly, we need to generate the overall terrain of the level before we can start generating other features. We will start with a random initial height, and on each tile, we will generate ground of the current height. While traversing the level, one or none of the following three events may occur:
 - *Increase or decrease current height* - if we want to have all the levels generated by the generator playable, we will need to constraint this height change to be at most the height Mario can jump (which is 4 tiles),

- *Generate gap* - if this event occurs, the generator will start a gap of a random length at the current position. The gap's length will need to be at most the distance Mario can jump,
2. **Pipes pass** - when we have the level's terrain generated, we can start generating other features. We will generate pipes of random height during this pass, but not more than Mario can jump. We will also need to forbid generating pipes at the places where are gaps or right after a gap or height increase because it may make the level impassable,
 3. **Bullet Bills pass** - this pass is almost identical to the previous one only instead of generating pipes we will generate bullet bills,
 4. **Stairs pass** - another feature that sometimes appears in Super Mario levels is stairs made of stone blocks so we decided to generate them too. The algorithm will randomly choose the length of the stairs. To not generate uselessly long stairs, we will constrain the maximum length to some reasonable number. The only thing we need to watch out for in this pass is to not generate them in places of gaps, or through other features since they are longer than one tile,
 5. **Platforms pass** - the last feature we will want to generate are platforms. These platforms are made of question mark blocks or brick blocks placed next to each other multiple times. In Super Mario Bros game, they always appear 4 or 7 levels above ground, so we will use this constraint too. Each type will have its own probability. Like stairs, they can have any length, but we will constrain it by some reasonable constant. We will also need to watch out not to generate them through other features,
 6. **Enemies** - the final pass will generate enemies. Each enemy type will have its own probability to be generated.

A list of all required probabilities is displayed in Table 4.1, and the results of all the passes are illustrated in Figure 4.3.

Pass	Probabilities
Ground	2 (height change, gap)
Pipes	1 (pipe)
Bullet Bills	1 (bullet bill)
Stairs	1 (stairs)
Platforms	3 (single platform, double platform, powerup in platform block)
Enemies	5 (one for each type of enemy)

Table 4.1: The table containing all the probability parameters our Multipass level generator uses.

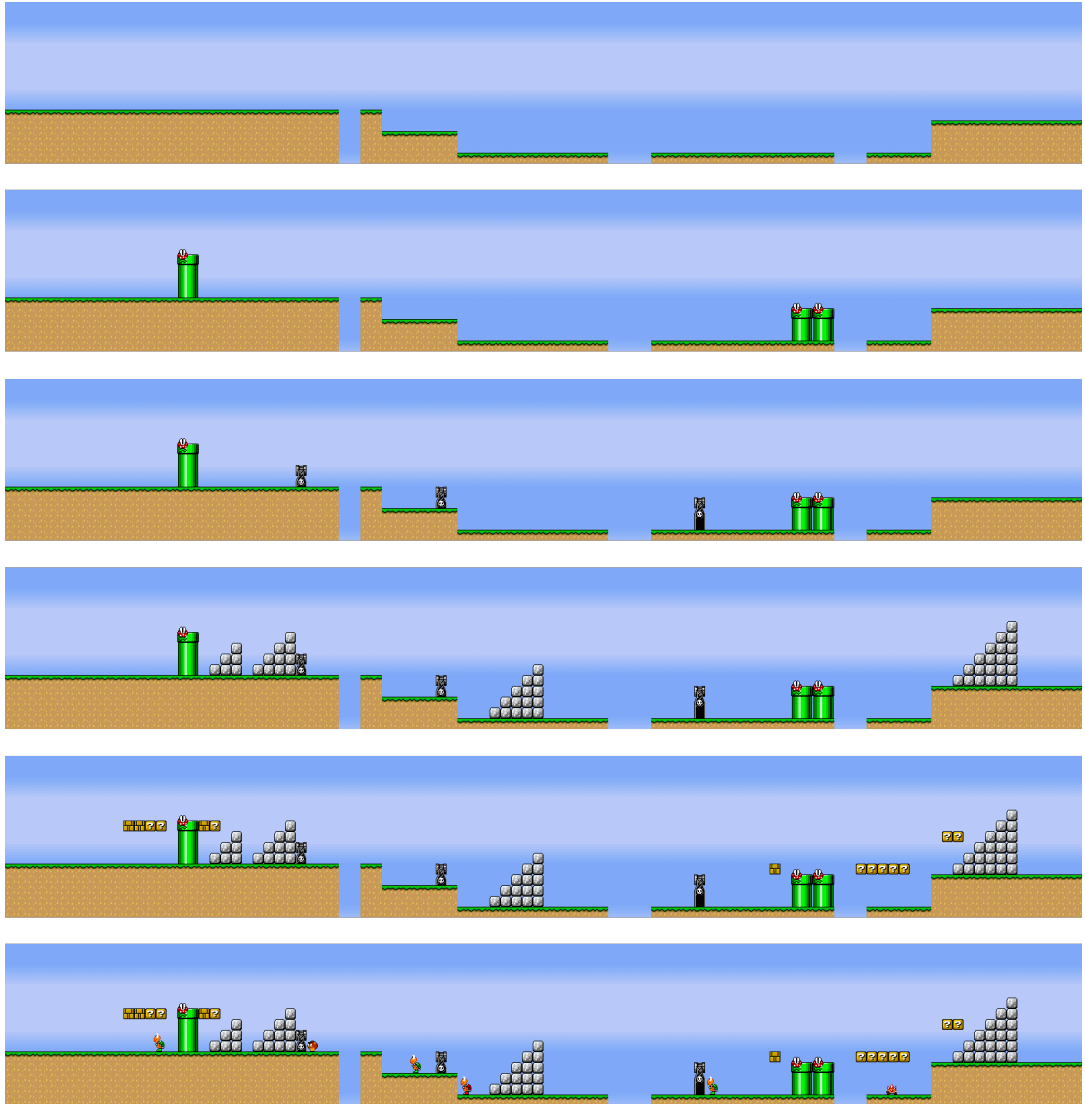


Figure 4.3: Visualisation of all level passes of our Multipass level generator. From top to bottom, each image displays the currently generated level after each pass.

4.2.3 Level postprocessing

We will also implement level postprocessing for purely aesthetic purposes. When generating levels, for simplicity, we were using only two types of ground tiles in both level generators: dirt tile and dirt tile with grass on the top. However, the tileset coming with Mario AI Platform contains also dirt tiles with grass on other sides to make the grass in levels continuous. To make our levels more visually appealing, we will implement level postprocessor, which will change border dirt tiles to dirt tiles with grass in the correct position. We will also generate some environmental tiles (e.g., arrow displaying direction of the level, plants) which have no impact on the level. An example of this postprocessing can be seen in Figure 4.4.

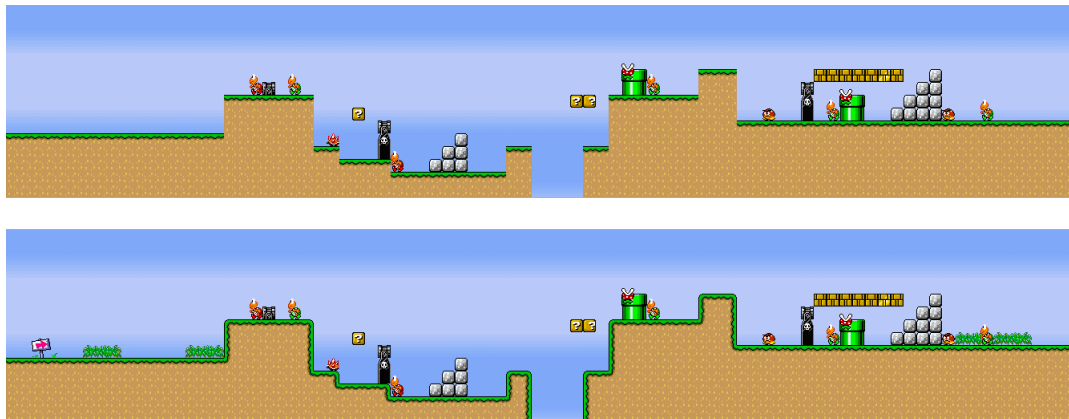


Figure 4.4: An example of level postprocessing. The original level is on the top and its postprocessed version on the bottom.

4.2.4 Evolution

Both our generators described above can be configured to generate a large variety of different kinds of levels. We will use evolutionary algorithms to find those configurations, which have properties we want.

Because the configurations are arrays of numbers (probabilities), we can use genetic algorithms for both. Our individuals will be fixed-size arrays of floating point numbers in the range $[0, 1]$. In the case of Chunked Markov Chain level generator, we will reshape the MC's transition matrix to a 1D array. We will run the evolution for a fixed number of generations using *gaussian* mutator, *elitism*, which preserves 2 best parents, and *roulette wheel selection*.

One thing we need to be careful with is the mutation in our Chunked Markov Chain level generator. We can not simply perturb probabilities in the individuals, because that would result in an invalid Markov chain. To solve this issue, we will implement a custom mutator for this evolution. It will do the following:

- **decrease a probability** - for each gene, with probability p decrease its value by a random value from interval $[0, 1]$. Then, coerce the resulting probability in $[0, 1]$, resulting in the final decrease by x . Finally, increase the values of all the other probabilities in the same row of the transition matrix (or the initial state vector) by $x/(n - 1)$ (n is the length of the row

or the initial state vector). This will preserve the constraint that each row of the transition matrix and initial state vector represent distributions of a random variable,

- **switching random variables** - to add more power to the mutator, we will also allow it to randomly switch transition matrix's rows.

4.2.5 Fitness function

Now that we have described how our evolution of level generators will look like, the only thing left is to define fitness function. We want our levels to be fun and interesting. The problem is that defining fun is not that easy. Instead of trying to do so, we propose to split this metric into multiple components. We will try to define these simpler metrics so that combined, they should measure how interesting the generated levels are.

We will want from our levels that their terrain won't be too flat, so we will need to define metric which measures their *linearity*. Next, we will want our levels to be challenging so that we will define a *difficulty* metric. The last thing we will want is, that the generators will generate all the possible features in a *complex* way. It would not be interesting if the levels would be empty, or contain only pipes, or some other feature. In the following sections, we will define these three metrics so that a computer will understand them, and we could use them in our fitness function.

Linearity

Terrain linearity is pretty straightforward to define. We can compute the average height change per tile at the given level and use it as this metric. Using this definition, a level with whose terrain relief is far from a line will actually have high *linearity metric* value.

Another way to compute this metric (and probably more common) is to calculate linear regression [39] (a straight line which approximates the level's terrain the most accurately). Then we can compute the distance of the actual terrain from the line. However, our metric also yields good results, as shown later.

Difficulty

Because difficulty is subjective metric (various players may be differently skilled, some may find a specific obstacle more difficult than others), we decided for the following approach. We assigned each type of obstacle a difficulty value and then sum them for each obstacle in the currently evaluated level. Table 4.2 displays the subjective difficulty values we assigned to each obstacle type and explanations why we decided on these values. As a starting point, we decided that normal *Goomba* will have difficulty value 1.

This approach is similar to *difficulty graphs* [40], which are sometimes used for difficulty evaluation.

Obstacle type	Difficulty	Explanation
Goomba	1	Pivot value
Green Koopa	2	Has 2 lives
Red Koopa	2	Has 2 lives
Spiky	3	More dangerous than Koopas since Mario will die when jumping on it
Flower	3	Basically the same mechanic as Spiky, only vertical
Green winged Koopa	4	Based on personal experience, these Koopas are pretty difficult. Even more difficult than Spiky
Bullet Bill	1.5 (height 1), 1 (other)	Bullet Bill is very similar to Goomba, but moves towards Mario. However, Bills flying above ground are not immediate threat.
Gap	0.5 (length 1), $0.5 + 0.5 *$ length (other)	Gaps of length x are basically x not moving Goombas in a row that cannot be jumped on. Gap of length 1 can be walked over on full speed
Platform	$-0.8 * \text{length}$	Platforms reduce difficulty in a way, that if enemies are spawned below them, Mario can just jump and walk on the platform without any threat

Table 4.2: Subjective difficulty values of all the obstacles occurring in the levels generated by our generators. Contains also brief explanation why we decided on the given values.

Complexity

The final metric we want to define is measuring the complexity of levels, meaning the diversity of appearing features. This is not an easy task on its own, but we propose the following solution. We will take an image of the level and compress it using a suitable compression algorithm, and the resulting image's size will be our complexity value.

Because compression algorithms can be pretty time consuming, computing this metric can have a significant impact on our evolution running time. To fight this issue, we will generate level images, where one tile takes space of 1x1 pixels (each tile using a different color) instead of 16x16, which will significantly improve compression time. An example of this minified image can be seen in Figure 4.5.

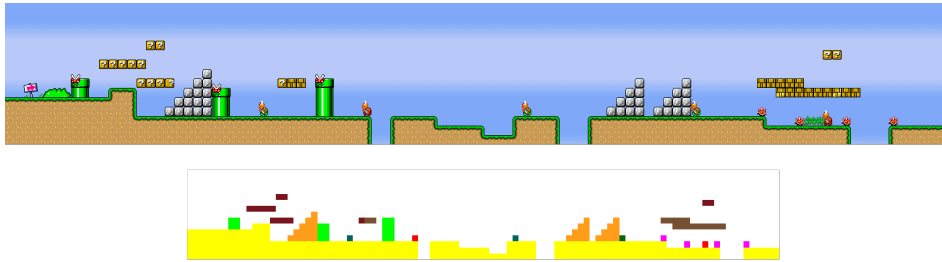


Figure 4.5: Illustration of our level image minification. Original image (top) is minified to a much smaller size (bottom, enlarged for better viewing), where one tile's size is 1x1 pixels. The minified image is used when computing complexity metric via an image compression algorithm.

The next thing we need to decide is which algorithm to use. We are looking for a compression, which takes advantage of repeating patterns. The more complex a level is, the more complex its image is, the less different patterns are repeated in the image, so the resulting size of the compressed image is higher. The PNG image format uses one such compression. It is called *DEFLATE* and uses a combination of *LZSS* [41] and *Huffman* compression. Both algorithms are particularly designed to look for repeating patterns, and the more such patterns they find, the more they reduce the size of the resulting image. Because of this property of these algorithms, we decided to use the size of the PNG image of a level as its complexity metric.

Fit for agent

The next property we will want from our generator is that it generates levels that are adequately difficult for the agent, so the agent can later improve by learning to solve them. We propose to achieve this by using fitness, which computes the agent's *win/loss* ratio on the current level generator. We will want to achieve a state where the agent solves 50% of these levels. This is inspired by machine learning techniques where half of the population are positive individuals and half negative ones to improve the learning process.

We can compute the difference between wins and losses on some number of levels and minimize this fitness. However, our previous metrics were designed to be maximized. Because we will be combining them, it would be more suitable to have this metric maximized too. Thanks to the fact that we will know the amount

of levels on which the generator will be evaluated, we can simply subtract the win/loss difference from that number. The resulting formula will be the following:

$$fitFitness = c - abs(2w - c)$$

where c is the number of levels on which the generator was evaluated, and w is the number of levels the agent solved.

4.3 Our results

We ran multiple experiments to verify whether we can achieve viable results by using evolution as specified previously. Firstly, we ran the evolutions while maximizing only one of the metrics at a time. By this approach, we wanted to find out whether our model is capable of evolving generators with the individual properties. In Figure 4.6, we can see a random level from each experiment. These images demonstrate that these experiments were successful for both types of generators.

After that, we tried to combine these metrics with the *fit for agent* metric. However, we excluded the difficulty metric because we were trying to make it fit the agent and not maximize it. The *fit for agent* metric is also the most important one. This is because we want to find out, whether the evolution can find level generators personalized for the player playing them. Because of this, we designed our combined fitness in the following way:

$$fitness = f * (1 + c + l)$$

where f is result from our *fit for agent* metric, c is result from our *complexity* metric, and l is result from our *linearity* metric. Both c and l metrics were normalized to the interval $[0, 1]$. To normalize linearity, we divided the original value by the length of the level and coerced it in the required interval. Finding maximum value of the complexity metric would be too difficult, so we run the evolution multiple times while maximizing that metric and used the maximum value the evolution found.

Using this fitness, we achieved that if the *fit for agent* metric is low, the resulting combined fitness is also low, so we force the evolution to primarily maximize this metric. A random level from some of the generators evolved using this fitness and various AI players for evaluation can be seen in Figure 4.7.

By running multiple experiments, we found out that population size of 50 can be evolved in 50 generations for preferred results. An evolution chart of one of the runs for each level generator can be seen in Figure 4.8. Looking at objective values in these charts, we can see that the evolution can find generators fit for the agent already in the first generations. More experiments can be found in the electronic attachment (Appendix A.1).

There is one difference between our generators evolved using *fit for agent* metric that stands out. Chunked Markov Chain level generator tends to put all the obstacles right at the beginning of levels, and the rest is just non-linear terrain. On the other hand, Multipass level generator places the obstacles more uniformly, which is expected from its definition.

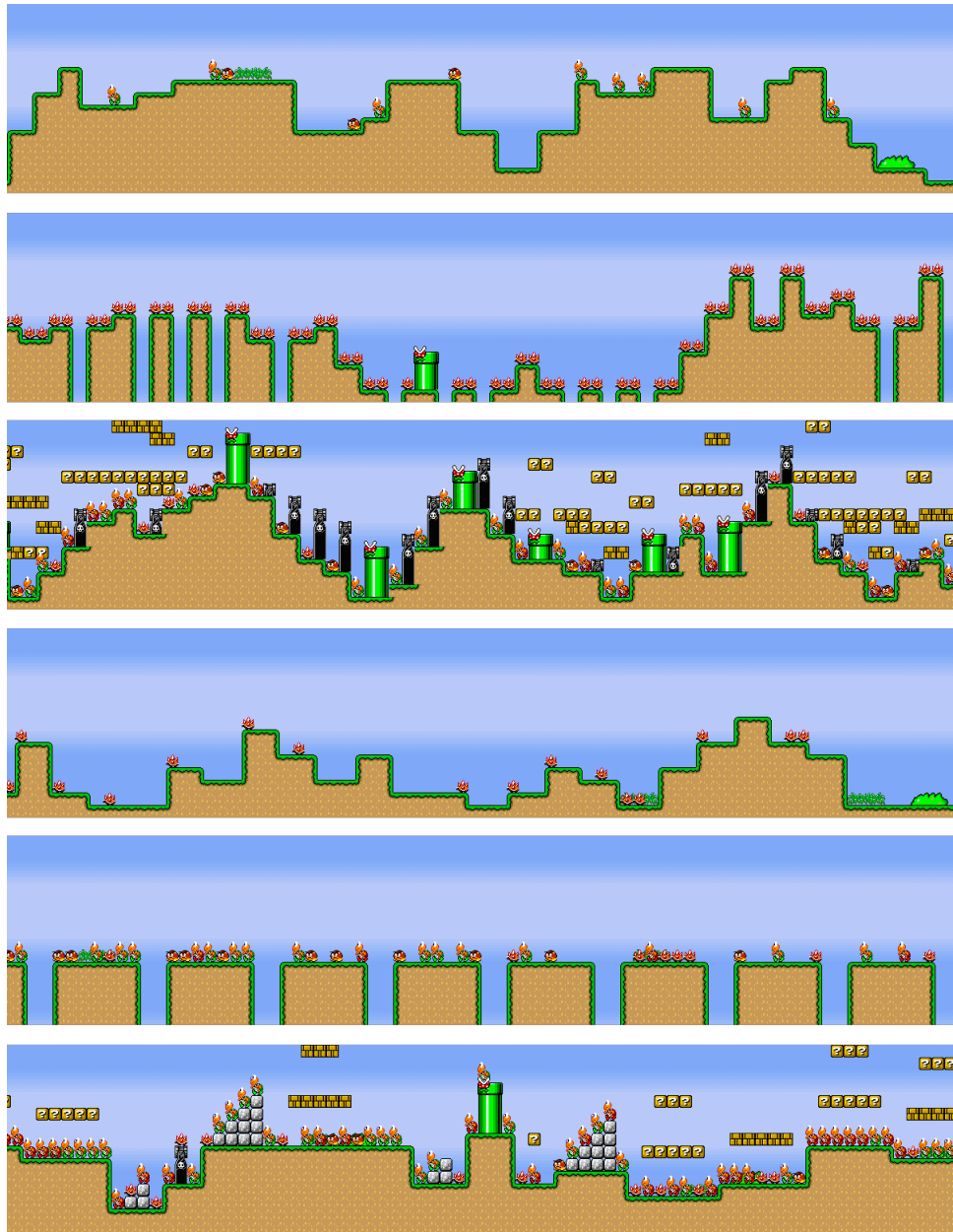


Figure 4.6: Examples of random levels generated by generators evolved using various fitness functions. From top to bottom:

- Multipass level generator - maximizing linearity metric,
- Multipass level generator - maximizing difficulty metric,
- Multipass level generator - maximizing complexity metric,
- Chunked Markov Chain level generator - maximizing linearity metric,
- Chunked Markov Chain level generator - maximizing difficulty metric,
- Chunked Markov Chain level generator - maximizing complexity metric.

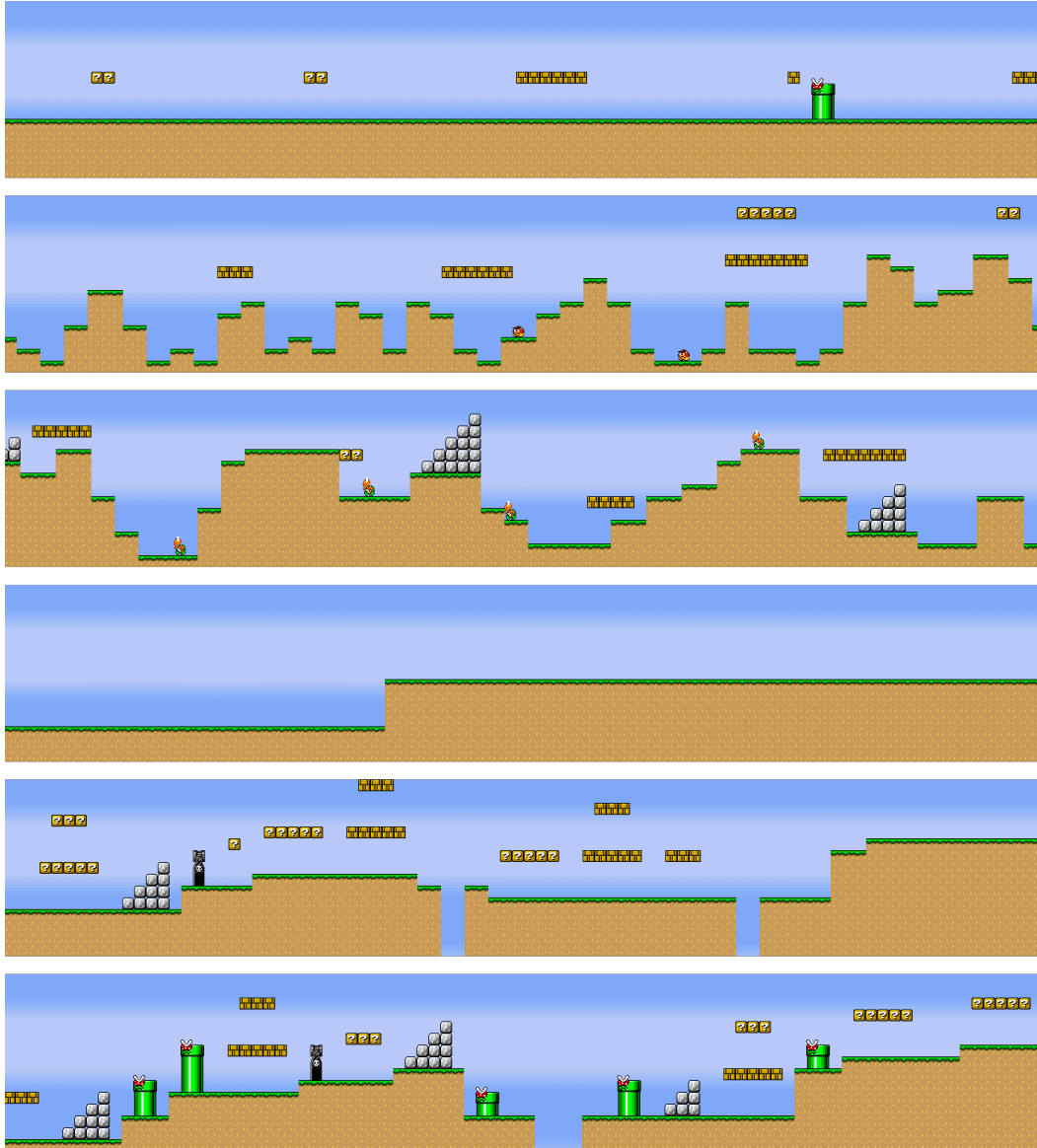


Figure 4.7: Examples of random levels generated by generators evolved using combination of *fit for agent*, *complexity* and *linearity* fitness functions and various AI players. When using an agent, which only goes right, the evolved generator generates flat levels with some obstacle, which appears only rarely, so that the agent solves 50% of the levels. When using an agent that goes to the right and jumps all the time, we can see that the evolved generator generates non-linear levels with occasional obstacles which can kill the agent. When using the best evolved NEAT agent, the evolved generator generates more complex levels. From top to bottom:

- Multipass level generator + going right agent,
- Multipass level generator + going right and jumping agent,
- Multipass level generator + best NEAT agent,
- Chunked Markov Chain level generator + going right agent,
- Chunked Markov Chain level generator + going right and jumping agent,
- Chunked Markov Chain level generator + best NEAT agent.

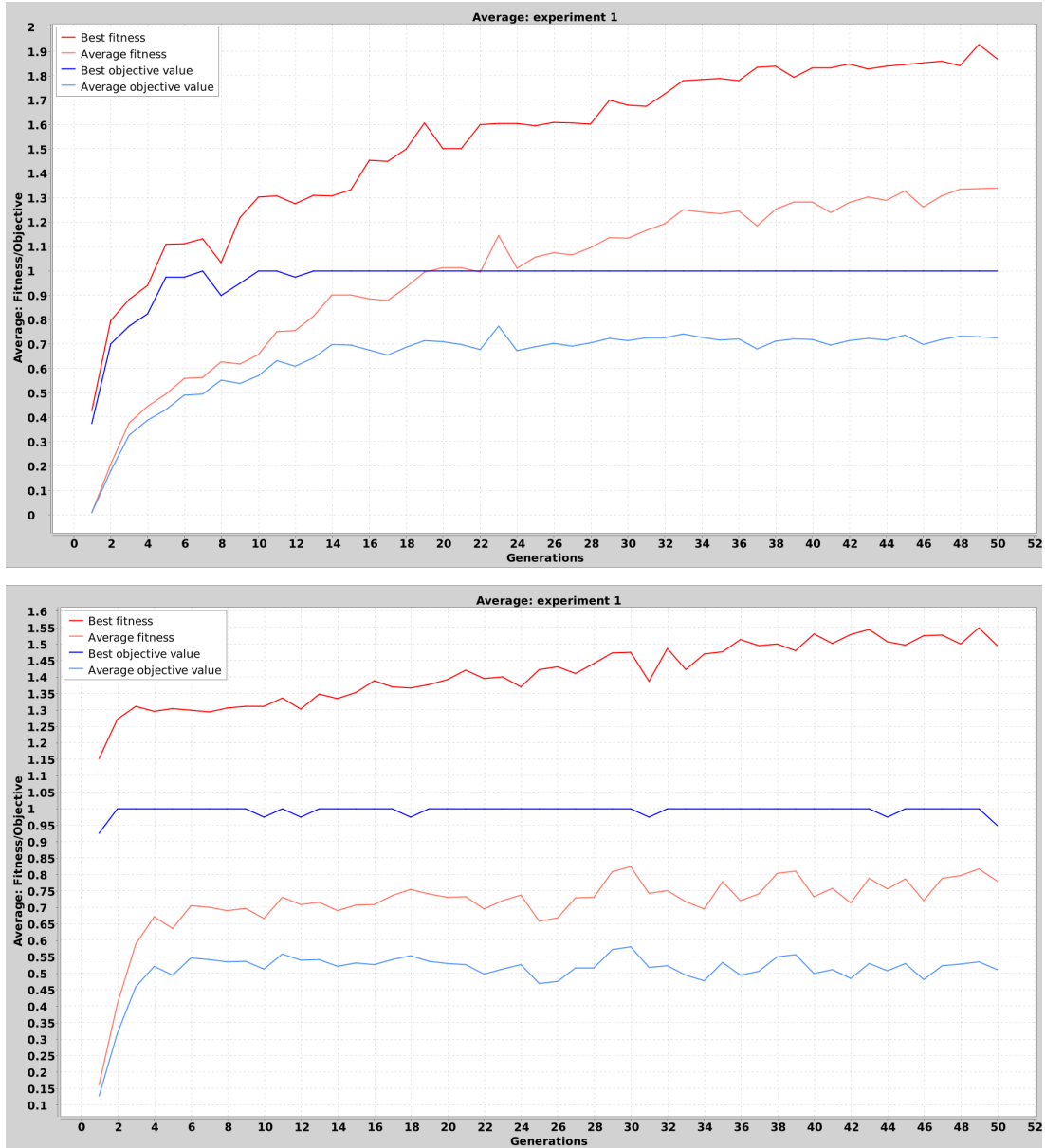


Figure 4.8: Charts of the evolution of Multipass level generator (top) and Chunked Markov Chain level generator (bottom). Both charts are averaged over 4 runs and use fitness, which combines *linearity*, *complexity* and *fit for agent* metrics using the best evolved NEAT player. The objective function is only *fit for agent* metric. Each metric's maximal value is 1. We can see, that even though the best evolved level generator did not maximize all of them (the fitness values do not reach value 3), it maximized *fit for agent* metric which we deemed to be the most important.

4.4 Conclusion

We have implemented two level generators, one using a multipass approach and the second one building blocks approach with Markov chains. Afterward, we used genetic algorithms to maximize the following properties of the levels they generate: *linearity*, *difficulty*, *complexity* and *fit for agent*. *Linearity* was computed as the average height change in the level, *difficulty* was calculated by summing up our subjective difficulty values of individual obstacles in the level, *complexity* by the size of compressed image of the level and *fit for agent* by win/loss ratio on multiple levels.

By running multiple experiments, we first proved that the evolution can maximize each of these metrics individually. After that, we were successful in maximizing the combined metric of these too, on both level generators.

There was one problem with Chunked Markov Chain level generator, and that was that it generates levels, where the obstacles are only at the beginning. However, during the following coevolution, it will be forced to generate more and more challenging levels, so it may be able to force it to generate obstacles throughout whole levels.

Because of these results, we find both generators to be good candidates for the later coevolution.

5. Coevolution of AI and level generators

After having the evolution of AI players and level generators for Super Mario implemented (chapters 3 and 4), we can start with the coevolution. We will start by examining some previous works, and then we will describe our approach. After that, we present our results and draw conclusions from this part of our work.

5.1 Related work

Similar work to ours was done by using coevolution to evolve endless runner type games [42]. It is a subgenre of platformer games, where the player is constantly running. The work was successful in generating new game rules and level generators and AI players for the generated game. It was initial research to coevolution usage in such a way. Our work differs from this in that it focuses more on competition between AI players and PCG content and explores this field in more depth.

Other work described POET (Paired Open-Ended Trailblazer) algorithm [43]. Even though it was not used for a computer game, it is very similar to our setting because it coevolved population of AI players and a 2D environment. The coevolution here is a little specific in that it evolved pairs of one player and one environment and occasionally swapped individuals between two pairs. Similarly, like we will be doing, they tried to evolve environments, which are adequately difficult for its paired AI player and then evolve the player to solve the new environment. After multiple runs, they took some of the evolved environments and tried to evolve AI players using the same evolution as in POET directly on these environments. They found out that it was unable to find players which would solve them as POET could.

In a different work, the authors used competitive coevolutionary techniques on the existing *SANE* algorithm, which evolves Go players [44]. In their algorithm, they coevolved two populations of AI players. The results of their work were that the coevolution evolved “better game-playing behavior” than standard evolution.

In other work, authors used cooperative and competitive coevolution to evolve AI player for Chinese Chess [45]. This work shows, again, that coevolved players perform “relatively well”. They also show a difference in results when using competitive vs. cooperative coevolution.

Different type of coevolution called *cultural coevolution* was developed to evolve AI players for game Go [46]. In this coevolution, only one population is being evolved, and the other (culture) contains the best individuals from some points of the evolution. Despite the fact that when faced against three benchmark agents, the simple evolution evolved agents performed better, when faced in tournaments against each other, coevolved agents won more tournaments.

There are also some works where the term coevolution was used when an EA algorithm evolved a single population, and the individuals of the same generation competed against each other [47] [48].

Except for the coevolution of endless runner games and POET, all these works

have the same trait: whether using one or more populations, they evolve only AI players. In the following sections, we will describe how we used the coevolution of populations of AI players and level generators for the Super Mario game.

5.2 Our approach

As decided in Chapter 2, we will be using *competitive coevolution*. It will evolve two populations: population of AI players and population of level generators.

At one time, only one of the populations will be evolving. While evolving population of AI players, we will use the best level generator from its population for evaluation, and vice versa, when evolving population of level generators, we will use the best AI player from its population for evaluation.

Now we must decide which population will be evolved first. If we want to start with the evolution of level generators, we would need to provide some AI player for evaluation, which we do not have. Randomly initialized AI player would probably only stand and do no actions. Because of this, it is impossible for the evolution of level generators to evolve such individuals upon which the player would have a 0.5 win/loss ratio (Section 4.2.5). On the other hand, if we start by evolving AI players, thanks to the design of our level generators, we can easily initialize both of them to generate only flat levels, which is a perfect starting point for evolving AI players.

After the evolution of AI players population finishes, we start the evolution of the level generators population, using the best evolved AI player for their evaluation. When this evolution finishes, we start our AI players evolution again, using the final population of the evolution's previous run as initial population of this run, and the best evolved level generator for evaluation. This process is repeated for some number of generations and is illustrated in Figure 5.1.

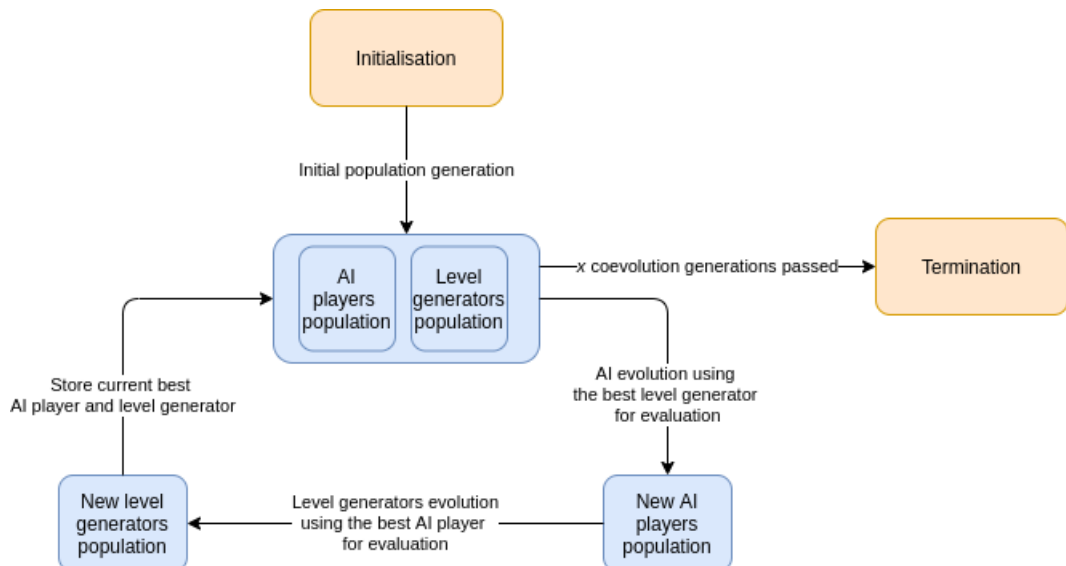


Figure 5.1: Our coevolution scheme.

When the coevolution finishes after x generations, we have a sequence of x AI players, which should be better and better, and a sequence of x level generator, which should generate more and more difficult and complex levels.

5.2.1 Encountered issues

After some experiments, we found a few shortcomings in our approach, which we tried to eliminate:

- **Difficulty cycling** - the first problem that arose was that the evolutions of level generators were not evolving generators which generated more and more difficult levels, but only “cycled” different obstacles. The best generator from the first generation learned to generate one type of obstacles. After that, the AI players learned to overcome them. Then, the next level generator learned to generate a different kind of obstacles, and the AI players learned to overcome them again but forgot how to overcome the first kind of obstacles. So, the next level generator learned to generate them again instead of the previous one. This created an infinite cycle, where the level generators were not generating more difficult levels gradually. To counter this issue, we have implemented a *sliding window* of level generators, which contained 5 latest best level generators, and the AI players were evaluated on these, instead of only on the last one, similarly as in the *cultural coevolution*,
- **Non-stability of AI player evaluations** - although evolved AI players had approximately 0.5 win/loss ratio on the best evolved level generators, when we rerun the evaluations again after the coevolution, we found a pretty significant deviation from this value. This was probably caused by the random behavior of our level generators, so increasing the length and number of the levels on which the players are evaluated during the evolution helped counter this issue,
- **Difficulty in multipass level generator** - the evolution of Multipass level generator tended to evolve generators, which generated levels consisting of one long brick platform with many enemies below it. The agent’s performance then depended only on whether it was able to get on top of the platform at the beginning of the level. Similarly, some evolved generators generated only stairs next to each other, which causes a similar problem. Some such levels are depicted in Figure 5.2. To resolve this issue, we constrained the generator not to be able to generate platform, stairs and many enemies directly next to each other.

5.3 Chosen algorithms

In Chapter 3, we saw that the NEAT algorithm produced better-performing players than neuroevolution of weights, but ran longer. For that reason, we will run the coevolution using both algorithms, but when using NEAT, we will run it with a smaller number of coevolution generations.

Later in Chapter 4, we saw that EAs of both our level generator produced similar results, so we will use both of them too, in case that some will yield better results in the coevolution.

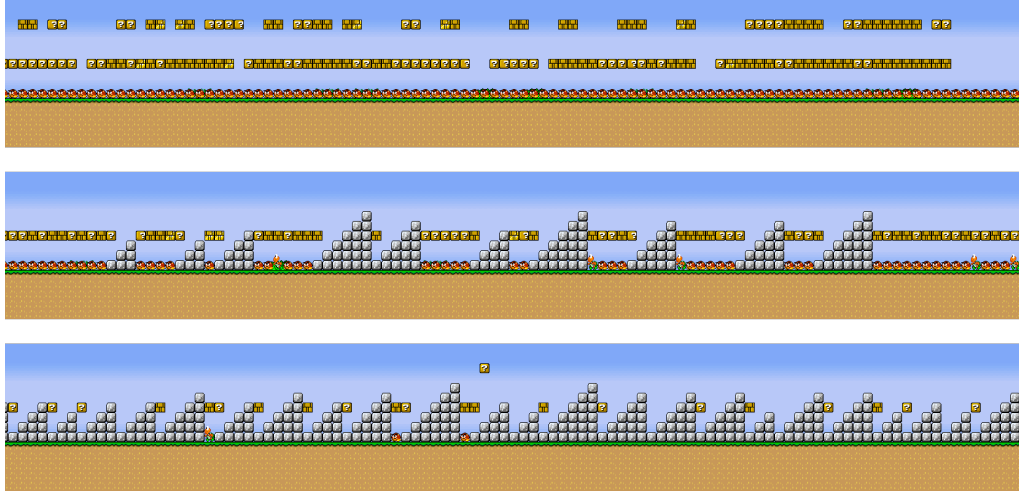


Figure 5.2: An illustration of difficulty and complexity metrics not mirroring the actual state in levels generated by Multipass level generator. Even though their values are high, the levels are not really difficult neither complex.

5.4 Our results

We ran the coevolution using all four possible combinations of our evolutionary algorithms. Each of these experiments was executed 10 times to get more relevant results.

5.4.1 Experiments using neuroevolution of weights

When using the neuroevolution of weights algorithm for AI player evolution, we let the coevolution run for 20 generations. The averaged coevolution charts of objective values from 10 runs can be seen in Figure 5.3.

From these, we can see that the coevolution is pretty successful in finding level generators that generate levels of adequate difficulty for a given player. The objective value for a level generator is 50000 when the player scores 0.5 *win/loss ratio* on its levels (evaluated on 30 levels). The value drops linearly to 0 when the ratio drops to 0 or jumps to 1 . The charts show that throughout the whole coevolution, the best individual from the level generators population scores maximal objective value after each level generator evolution run.

The objective value of AI players is computed as a number of solved levels times 1000 . The maximum value is therefore 25000 because the players were evaluated on 25 levels. We can see that only in the first generation of the coevolution can the best players learn to solve all the levels (which are only flat levels with no obstacles). Afterward, only in a few following generations the players are able to learn to overcome the new obstacles (the maximal objective value of AI players is increasing during one evolution run). This is more apparent when Multipass level generator was used (Figure 5.3).

A video showcasing one of the coevolution runs can be found in the electronic attachment¹ (see A.1).

¹Also at <https://www.youtube.com/watch?v=bthgxZYPaCs>

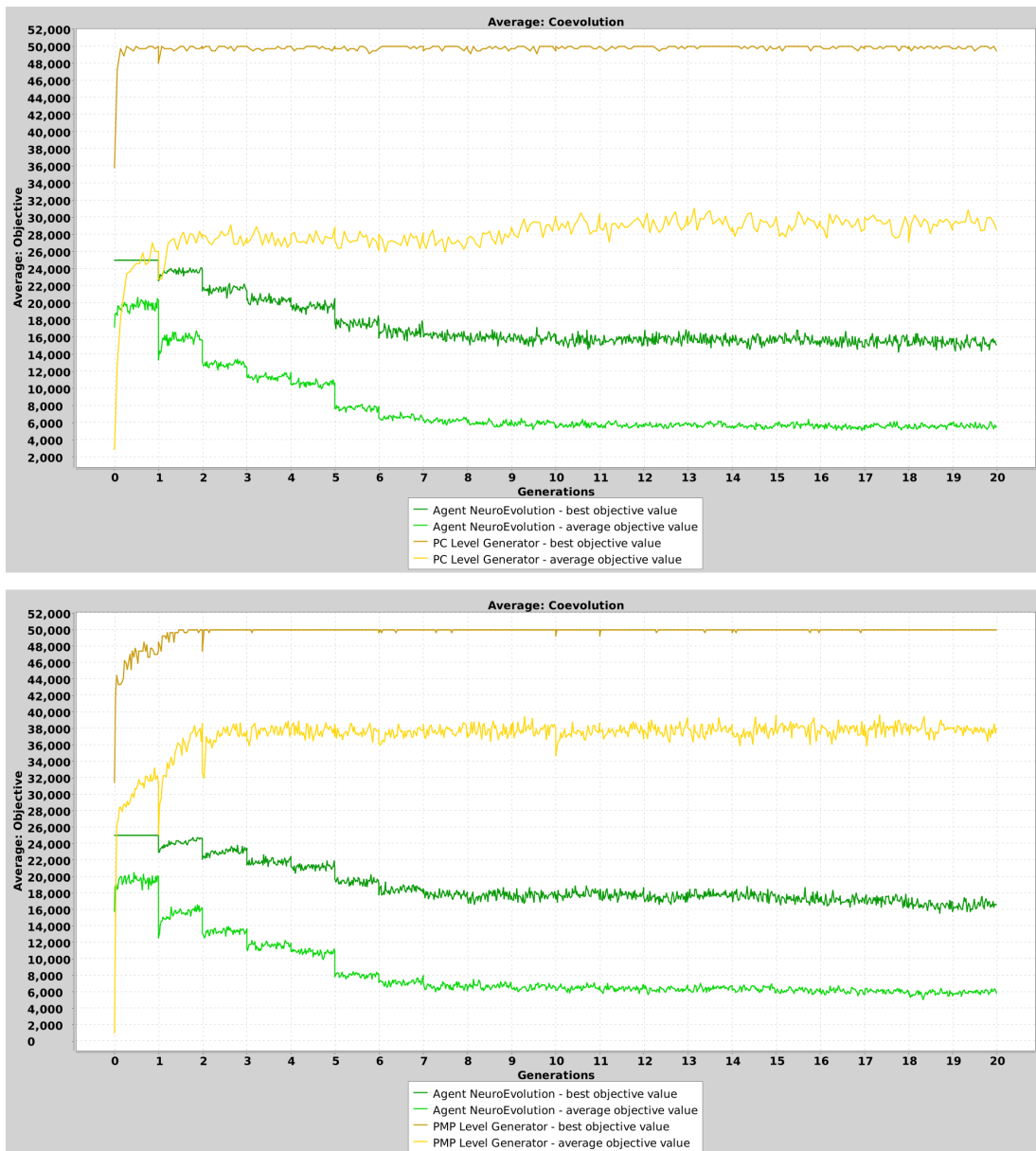


Figure 5.3: Charts of objective values averaged over 10 coevolution runs. The *x-axis* on both charts contains single evolution generation numbers normalized so that one unit represents one evolution run. In other words, it contains coevolution generation numbers. The *y-axis* contains objective values (*fit for agent* metric). The neuroevolution of weights evolved the AI player. The level generators used were Chunked Markov Chain level generator (top) and Multipass level generator (bottom). From the objective values of level generators (*fit for agent* metric), we can see that the coevolution easily finds level generators which generate adequate levels for the players. We can also see that during the first runs of AI evolution, it is able to learn players to overcome the new, more difficult levels.

5.4.2 Experiments using NEAT

When using NEAT for AI player evolution, we run only 10 generations of coevolution. We also reduced the number of generations in the evolution to 400 (down from 500) to fight the higher run time of NEAT. The averaged coevolution charts of objective values from 10 runs can be seen in Figure 5.4.

Similarly to experiments with the neuroevolution of weights, the evolutions of level generators can easily find required level generators. The objective values of NEAT individuals, on the other hand, do not increase at all except in the very first run. Upon observing the agents, they do not seem to learn more than going right and jumping. This is an interesting result because NEAT evolved better agents than the neuroevolution of weights when trained on a fixed set of levels (Chapter 3). This may be caused by NEAT overfitting on that training set and being unable to generalize on PCG generated levels.

5.4.3 Running times

Most of the coevolution experiments were run on a machine with *Intel® Core™ i7-6700* processor which has 4 cores running at 3.40GHz and uses *hyperthreading*, 16GB memory and *Gentoo 2.6* operating system. The average run times can be seen in Table 5.1.

5.5 Evaluation of AI players

To evaluate whether our players evolved by coevolution perform better than players evolved by an evolution, we will execute the following experiment. We will take some of the level generators evolved by coevolution and let our EAs evolve players using this generator. In the end, we will compare how many levels that evolved player solved with how many levels the player evolved by coevolution solved. A similar evaluation of coevolution evolved players was done with the POET algorithm [43].

We chose five of the final level generators evolved by our coevolution, and run the *neuroevolution of weights* using this generator. We did not run this experiment with NEAT players because they did not seem to perform well.

We let the evolutions run for 150 generations (3 times more than we used for training data set in Chapter 3). After that, we took the evolved agent and coevolved agent from the same experiment and let them run 200 levels generated by the generator.

In each of these experiments, the coevolved agent solved more levels than the evolved one, as can be seen in Table 5.2. However, the difference does not seem to be as significant as for the POET algorithm [43], for example. The evolution chart of the first experiment is depicted in Figure 5.5.

5.6 Evaluation of level generators

To evaluate whether the level generators are improving between generations of the coevolution, we plotted our individual metrics on each generation's best level

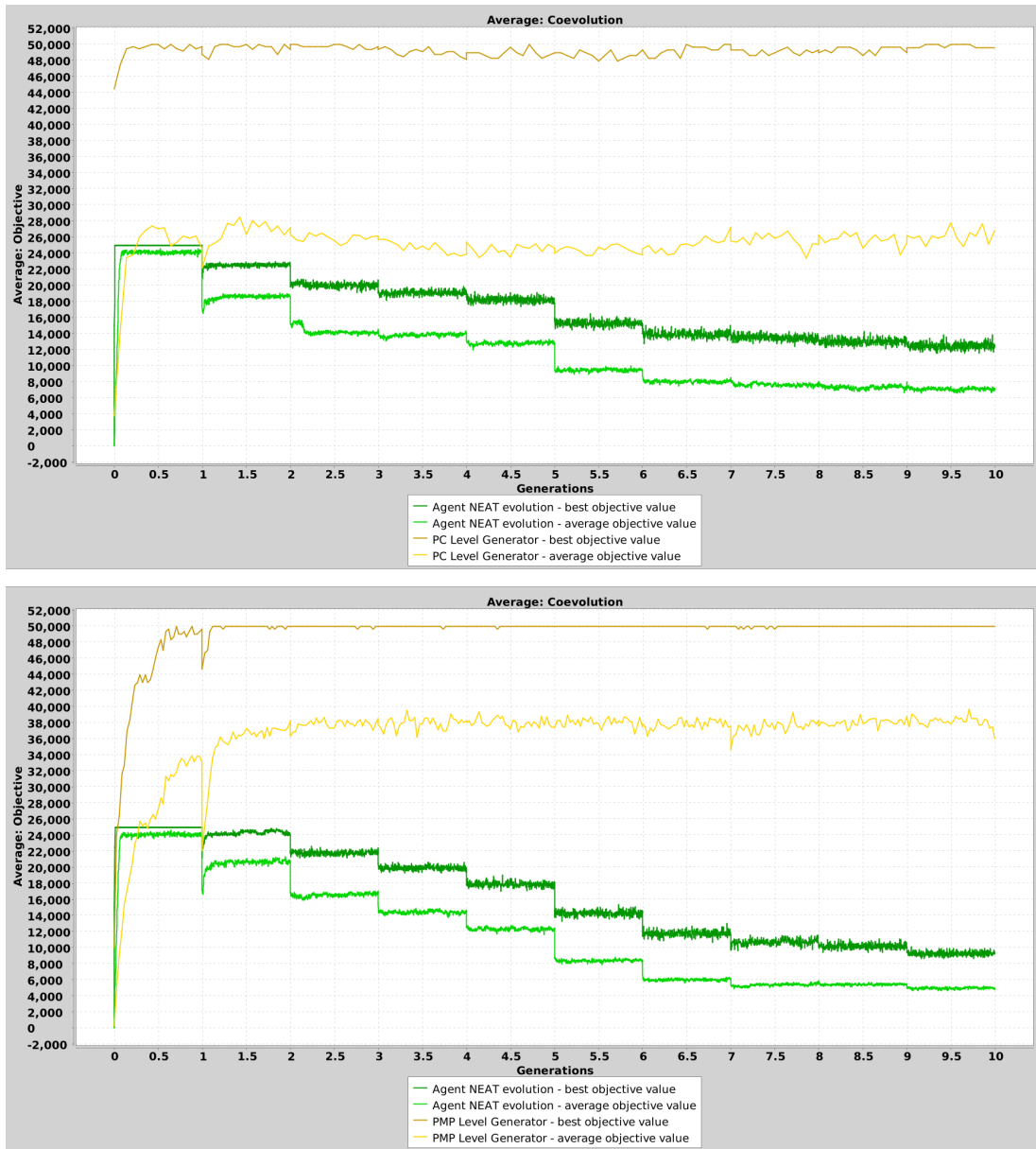


Figure 5.4: Charts of objective values averaged over 10 coevolution runs. The *x-axis* on both charts contains single evolution generation numbers normalized so that one unit represents one evolution run. In other words, it contains coevolution generation numbers. The *y-axis* contains objective values (*fit for agent metric*). NEAT evolved the AI player. The level generators used were Chunked Markov Chain level generator (top) and Multipass level generator (bottom). From the objective values of level generators, we can see that the coevolution easily finds level generators which generate adequate levels for the players.

Algorithms	Total time	AI evolution time	LG evolution time
NoW + Chunked	8:42	6:46	1:57
NoW + Multipass	14:05	7:57	6:08
NEAT + Chunked	14:40	14:09	0:31
NEAT + Multipass	30:30	28:34	1:55

Table 5.1: Average running times of coevolution experiments. The times are in hours. *NoW* stands for neuroevolution of weights, *Chunked* for evolution of Chunked Markov Chain level generator and *Multipass* evolution of Multipass level generator.

Coevolution experiment	Levels solved by coevolved agents	Levels solved by evolved agent
Multipass - 5	33%	7%
Multipass - 8	49%	48%
Multipass - 10	49%	37%
Chunked - 3	36%	34%
Chunked - 7	51%	45%

Table 5.2: Comparison of coevolution evolved (using either Multipass or Chunked Markov Chain level generator) and the neuroevolution of weights evolved agents. The level generators used are taken from the results of picked coevolution experiments. Each player played 200 levels.

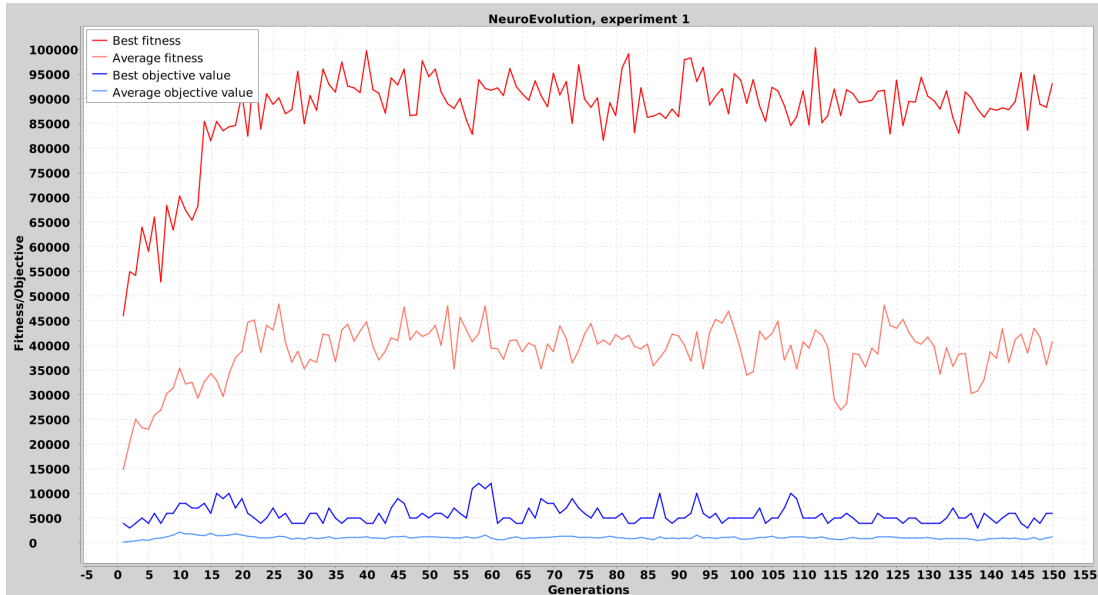


Figure 5.5: Evolution chart of *neuroevolution of weights* using the final level generator from one of the coevolution experiments. The player was evaluated on 20 levels during the evolution. The fitness function is the sum of distances the player reached in all the levels and objective function is the number of levels it solved times 1000.

generator. Because we ran 10 experiments for each coevolution, we averaged results from all of them.

The charts showed that *linearity* and *complexity* were increasing during the coevolution. It even seems that for Multipass level generator, they are not yet converged and that they would increase for a few more generations. The difficulty of levels generated by *Multipass level generator* seem to be increasing at least for the few generations, where the performance of AI player was increasing too. All of this can be seen in Figure 5.6.

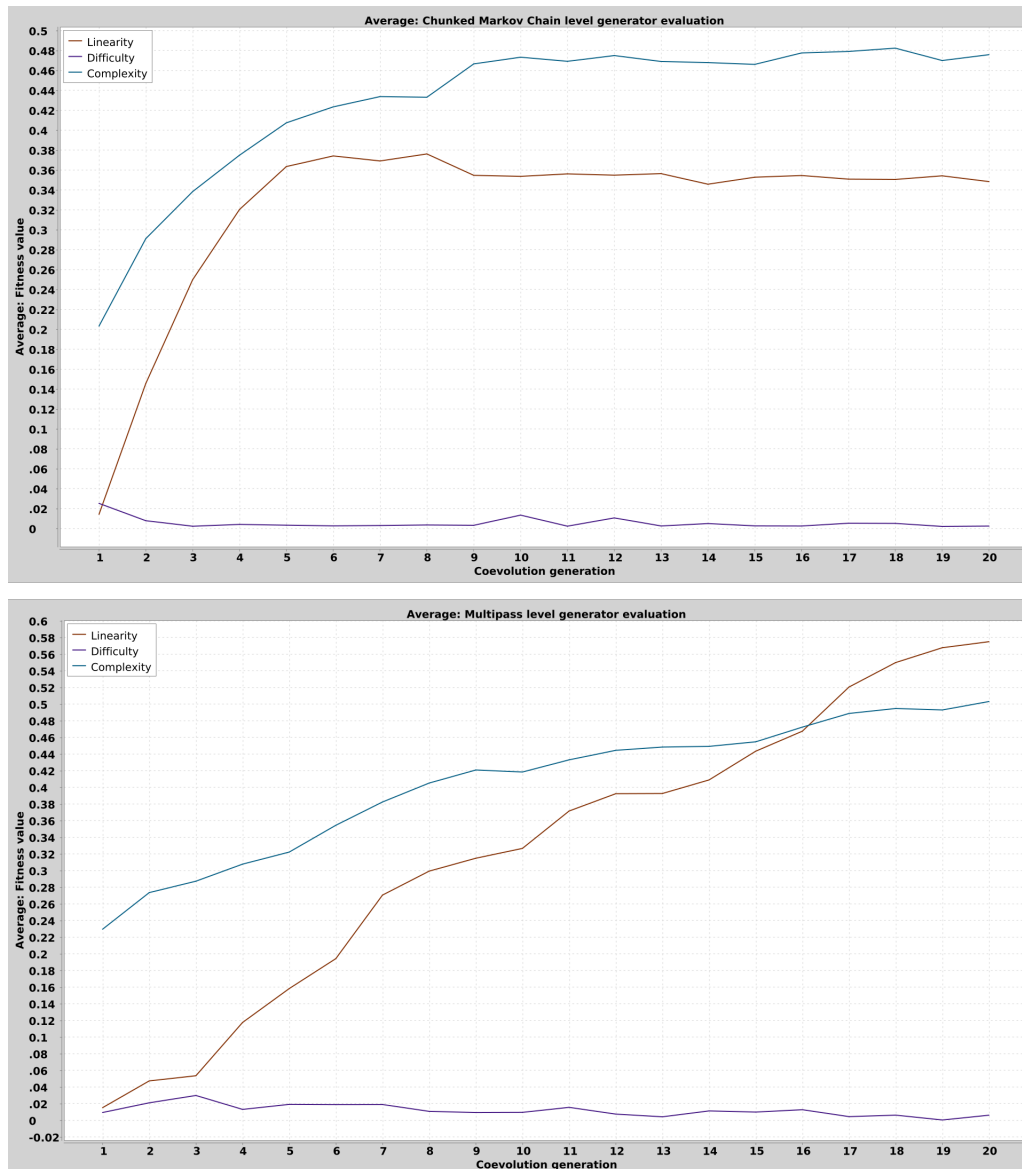


Figure 5.6: Evaluation chart of evolution of Chunked Markov Chain level generator (top) and Muultipass level generator (bottom). The *x-axis* contains coevolution generation. The *y-axis* contains fitness values of the best evolved level generator from a given generation, averaged over 50 levels it generated. The difficulty is not increasing much, while complexity and linearity are. For *Multipass level generator*, it even seems that it would increase in a few future generations.

5.7 Conclusion

In this chapter, we have implemented the coevolution of populations of AI players and level generators for the Super Mario game. For AI players evolution, we have used the neuroevolution of weights and NEAT algorithms. For level generators, we have used evolutions of Chunked Markov Chain level generators and Multipass level generators.

After implementing the coevolution, we have run experiments using all four possible combinations of these algorithms. Each experiment ran multiple hours, was run 10 times, and their results were averaged.

From the coevolution charts, we could see that evolutions of level generators were pretty easily able to find adequate level generators for the AI players on which they were evaluated. The evolutions of AI players, on the other hand, were not always able to find agents which would solve all the generated levels. Specifically, they were able to find such only in the very first generations of the coevolution.

Still, because the AI players' performance improved at least slightly in the coevolution generations, we got our sequence of level generators, whose generated levels' difficulty is gradually increasing.

6. Implementation

In this chapter, we will describe our implementation of this work in more detail. Firstly, we will outline the code structure, then we will describe important parts of the code and finally we will look at the code documentation.

The whole codebase can be found either in the electronic attachment (see Appendix A.1) or as a git repository on GitHub¹.

6.1 Code organisation

As decided in Chapter 2, all of our code is written in Kotlin language. The only parts of the codebase written in Java are two libraries which are not downloaded automatically by our build tool, *Gradle*. One of them is *Mario AI Framework* and the second one is *NEAT*. The reason why they are not automatically downloaded is that we made a few changes in them, so they need to be recompiled during the build process.

The project is split into two subprojects. One containing the *Mario AI Framework* (folder *MarioAI4J*) and the second one is our project (folder *MarioDoubleEvolution*). *MarioDoubleEvolution* subproject follows Gradle's standard directory structure. The sources are split into two directories: *src/main* which contains implementation of the functionality and *src/test* containing unit tests. Source files in these two folders are then split into subdirectories by the language in which they are written. The only Java code in this subproject is the NEAT algorithm, located in *src/main/java*. All the other code is in *src/main/kotlin*. Resource files like tilesheets are located in *src/main/resources*.

All of our packages start with prefix `cz.cuni.mff.aspect`. This package is split into multiple child packages, which we will later refer to as root packages:

- `coevolution` - contains implementation of our coevolution from Chapter 5,
- `controllers` - contains implementation of a few simple AI players used for benchmarking,
- `evolution` - contains implementation of all our evolutions mentioned in Chapters 3 and 4,
- `launch` - contains multiple entry points. They are all described in Appendix B in more detail,
- `mario` - contains integration between our codebase and *Mario AI Framework*,
- `storage` - contains implementation of storing JVM objects to binary files and text to text files on the disk,
- `utils` - contains utility functions and extension methods used in multiple places in the project,
- `visualisation` - contains implementation of various visualisations.

¹<https://github.com/Aspect26/MarioDoubleEvolution>

6.2 Mario AI Framework integration

The integration with *Mario AI Framework* is implemented in the root package `mario`. It contains class `GameSimulator`, which provides multiple methods to run *Super Mario* game using specified AI player and levels. Levels are represented by our `MarioLevel` interface, and AI players implement interface `MarioController`.

Mario AI Framework uses its own level generator for generating levels in its simulator. Because we needed to tell the simulator to play a concrete level, we implemented `SingleLevelLevelGenerator`, which gets an instance of `tMarioLevel`, converts it to level representation used in the framework and returns it when asked from the simulator to generate new level.

6.3 Evolutions

All evolutionary algorithms are implemented in the `evolution` root package. It is split into multiple subpackages, notably: `controller` for AI player EAs and `levels` for level generator EAs. To easily switch different EAs in our coevolution, we created interfaces `ControllerEvolution` which every EA of AI player must implement and `LevelGeneratorEvolution` which every EA of level generators must implement. Additionally, each EA except NEAT is implemented using *jenetics* library, so all the common code is extracted to `ChartedJeneticsEvolution` abstract class, which also contains implementation of plotting the evolution to a line chart. This whole hierarchy can be seen in Figure 6.1.

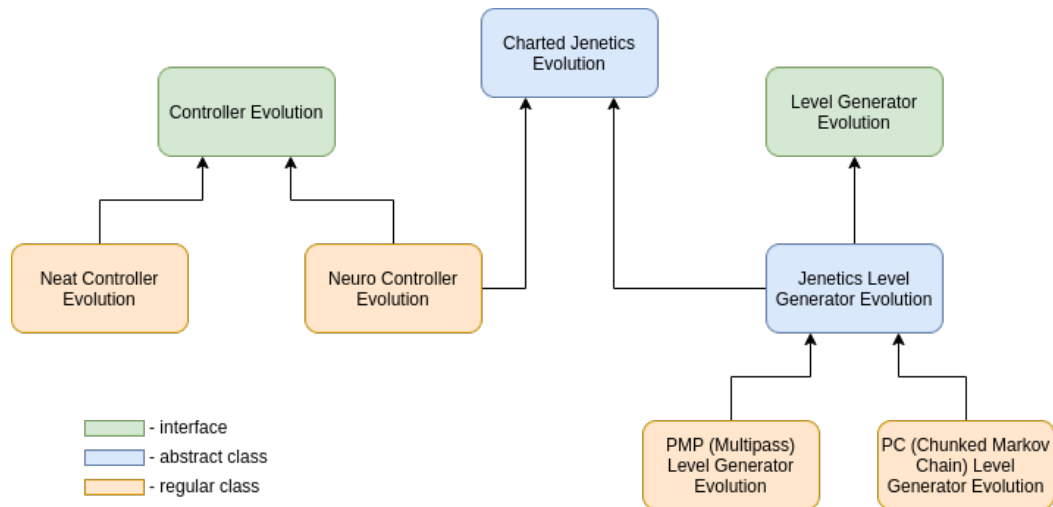


Figure 6.1: Hierarchy of our classes representing EAs. Arrows are pointing from a child class to a class from which it inherits (or interface it implements). There are inserted white space after each word in the class/interface name for better readability.

6.4 Coevolution

Our coevolution algorithm is implemented in the `coevolution` root package, specifically in `Coevolution` class. As an input, the algorithm takes an object of `CoevolutionSettings` type, which specifies particularly which EAs should

the coevolution use, the initial level generator and AI player and for how many generations the coevolution should run.

In case a long running coevolution should crash or stop for any reason, we have implemented a way to restore it. The state of coevolution, represented by an instance of `CoevolutionState` class, is stored on a disk after each single evolution is finished. Thanks to this, using the same `CoevolutionSettings` object that was used to start the coevolution it can be restarted from this point. This will result in losing only the last non-finished single evolution progress, which does not take a large amount of time.

A `CoevolutionResult` object is returned after the coevolution is finished, which contains the best evolved controller and level generator. The best individuals from each single evolution's generation are stored on the disk. These can be loaded via `ObjectStorage` to retrieve our sequence of level generators of increasing difficulty.

6.5 Visualisations

In our project, we have implemented two kinds of visualisations. One displays image of a *Super Mario* level and the other displays linechart of an evolution.

6.5.1 Level visualiser

We have implemented `LevelToImageConverter` object, which takes as an input a `MarioLevel` object and creates an image of the level. `MarioLevel` object specifies exactly what tiles are at which positions in the level, as well as all the entities. In combination with using tilesheets from *Mario AI Framework* it creates the resulting image of the input level². The converter can create also minified image of a level as specified in Section 4.2.5.

`LevelVisualiser` then creates a GUI frame using Java's *Swing* library with a single component. This component draws the image received from the previously mentioned converter. One such visualisation can be seen in Figure 6.2

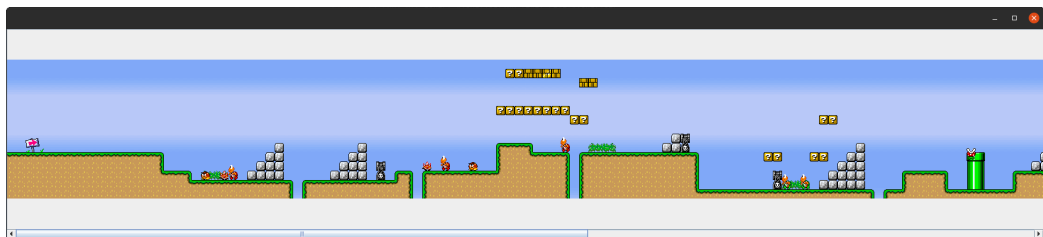


Figure 6.2: An example of a level visualisation using our level visualiser tool.

6.5.2 Evolution charts visualiser

To get a better insight into what is happening during our evolutions, we have implemented a visualiser which displays its state in real-time - `EvolutionLineChart`

²We rotated the enemy sprites around the vertical axis because it makes more sense for the entities to be left-facing (in direction of Mario's location).

class. Specifically, it displays the best fitness value, the average fitness value, the best objective value and the average objective value on *y-axis* of each generation (*x-axis*). Each of these values are displayed as one line in a linechart.

We have implemented also a different linechart visualisation for coevolution - `CoevolutionLineChart` class. It gets two `EvolutionLineChart` objects as an input, which it combines into two coevolution charts. One displays the average and the best fitness values of both evolutions and the other the average and the best objective values of both evolutions. It also changes the *x-axis* so that one unit on the axis corresponds to one evolution run.

All of our linecharts use `LineChart` class to display the charts, which uses `XCharts` library for the actual rendering. It implements also some additional functionalities, which are:

- **storing the data series**, thanks to which we can recreate and/or edit the charts later,
- easier **update** of the data series,
- **storing** the chart on the disk as an SVG image.

Finally, we have implemented `AverageLineChart`, which can take multiple `LineChart` objects as an input, matches series based on their labels and draws a new chart where the values from the input charts are averaged.

6.6 3rd party libraries changes

We needed to do a few changes in the 3rd party libraries that we use in our project. These changes affect NEAT library and Mario AI Framework, and they are described in the following sections.

6.6.1 NEAT library changes

The most notable changes we did in NEAT library are the following:

- **serialization** - in order to be able to store the coevolution state, we needed to store and load the last population of NEAT evolution. The library does not provide this functionality, so we had to implement it ourselves, as well as retrieving the population in the last generation,
- **configuration** - the configuration of the algorithm was originally in a static class, implemented as set of constants, meaning that it could not be changed, stored nor loaded. To resolve this issue, we made the parameters we use (input/output nodes count and population size) instance parameters,
- **population initialisation** - the initial population was always initialized to the initial NEAT individuals (Section 1.5.2). However, we needed to be able to initialize it to a specific population in some cases (e.g., when starting a next NEAT evolution in a coevolution), so we implemented also this possibility.

6.6.2 Mario AI Framework changes

The following changes were done to the *Mario AI Framework*:

- **gradle** - we've added *gradle* support to the framework, so that we can integrate it into our project more easily,
- **singleton environment** - the framework has an issue that we could not run multiple simulations at a time because some classes and members were *static* or implemented as a *singleton*. Because of this, we made static members instance members, and reimplemented the singleton *LevelGenerator* into a standard class,
- **interface LevelGenerator** - we also interfaced its level generator so that we can use our own implementation,
- **Mario's initial state** - the simulator starts with Mario being in Fire state (the state after picking up Fire Flower powerup). We changed this so that Mario starts in its initial state just like in the original game.

6.7 KDoc

More specific details about our implementation can be found in the code documentation in electronic attachment (see Appendix A.1), which is also published at the project's GitHub pages³. It was generated using *dokka* tool, which uses *KDoc* documentation comments in the code [49].

³<https://aspect26.github.io/MarioDoubleEvolution/-mario-double-evolution/>

Conclusion

In this work, we looked at a “competition” between *AI players* and *level generators* for classic game *Super Mario*. We implemented multiple *EAs* which evolve these. Afterward, we implemented the *competitive coevolution* of these two populations and measured improvements in results when using this approach instead of evolving each population individually.

To evolve AI players, we used the *neuroevolution of weights* and *NEAT* algorithms. To evaluate which agents performed better, we let them play through some of the original Super Mario Bros game levels. NEAT players were able to solve more levels, however; they took more time to evolve than neuroevolution of weights.

We also created two level generators. One of them was based on the *building blocks* principle combined with *Markov chains* (Chunked Markov Chain level generator). The other one was based on the *multipass approach* (Multipass level generator). Both of the generators were *probabilistic*, meaning that they generate levels based on given probabilities. We then used evolutionary algorithms to evolve these probabilities so that the levels they generate have properties we wanted. After various experiments, we found out that the generators’ expressive range allows them to generate levels that are highly non-linear, difficult or complex and also combination of these. Finally, we were also able to generate levels which are adequately difficult for a given AI player. However, Multipass level generator generated more appealing levels because Chunked Markov Chain level generator tended to generate all the obstacles right at the beginning and then just empty terrain.

Finally, we implemented coevolution where evolutions of populations of AI players and level generators take turns. Even though NEAT players performed better when being evolved on a fixed set of levels, when evolved by coevolution on randomly generated levels, they performed worse than those evolved by the neuroevolution of weights. This may have been caused by NEAT overfitting on the training data set or bad generalization. When evaluating our coevolution evolved players, we found out that it was able to find players, which solved more complicated levels than the neuroevolution of weights. This leads us to the conclusion that it may be easier to learn AI players step by step rather than let them learn directly on difficult levels.

Regarding level generators, we thought that the problem with our Chunked Markov Chain level generator would be solved by evolving better-performing players during coevolution, which would force it to fill up the levels with obstacles. However, the evolved players’ performance turns out to be insufficient to achieve this, so levels generated by Multipass level generator looked more natural (in a way, level generators were more successful at their task of presenting challenges to the players than the players were at their task of solving them).

Linearity and complexity of the best level generators from each coevolution generation were also visibly improving. This means that we were able to get a sequence of level generators, which gradually generated more and more interesting levels. However, the difficulty was not increasing that much. This was most probably caused by the fact that after some point, the evolution of AI players

was not able to find better players, which would solve more difficult levels.

Future work

There are various ways in which our work may be enhanced. One of them is implementing better AI player, which might be achieved through *deep reinforcement learning*, for example. This improvement is desirable because the evolution of level generators seemed to be more powerful than that of AI players. It could result in having more level generators of different difficulties as a result of our coevolution. If the learning technique would be more precise, we could even end up with finer difficulty steps in the level generators.

Another potential update to our work would be implementing a system for learning an AI player to mimic a human player’s behavior. If we would use this player in our evolution of level generators, we would end up with a level generator which would generate *personalized content* for the human player. Additionally, by using different win/loss ratios in our fitness, we could prepare a set of level generators of various difficulties tailored for the specific human player.

If we leave the world of Super Mario, we can still use the same evolutionary algorithm for AI player, since it is not bound to this game only. By replacing our level generator evolution with something more general, e.g., by using GVGAI platform [35], our coevolution would then be able to generate our sequence of level generators of gradually increasing difficulties for any game.

Bibliography

- [1] S.L. Kent. *The Ultimate History of Video Games: Volume Two: from Pong to Pokemon and beyond...the story behind the craze that touched our lives and changed the world*. Crown, 2010.
- [2] Sean Murray. Gamescom 2014: The sun will burn out before you see all of No Man’s Sky. <https://www.ign.com/articles/2014/08/15/gamescom-2014-the-sun-will-burn-out-before-you-see-all-of-no-mans-sky>. (accessed July 14, 2020).
- [3] Mark Hendriks, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Io-sup. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 9(1):1–22, 2013.
- [4] Max Freeman-Mills. Mike Booth, the architect of Left 4 Dead’s AI Director, explains why it’s so bloody good. <https://www.kotaku.co.uk/2018/11/19/mike-booth-the-architect-of-left-4-deads-ai-director-explain-why-its-so-bloody-good>. (accessed June 19, 2020).
- [5] Ian Millington and John Funge. *Artificial Intelligence for Games*. CRC Press, 2009.
- [6] Julian Togelius, Noor Shaker, Sergey Karakovskiy, and Georgios N Yannakakis. The Mario AI Championship 2009-2012. *AI Magazine*, 34(3):89–92, 2013.
- [7] Ahmed Khalifa, Sergey Karakovskiy, Noor Shaker, Julian Togelius, and Markus Persson. Mario AI Framework. <http://marioai.org/>. (accessed June 7, 2020).
- [8] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.
- [9] Kate Compton and Michael Mateas. Procedural level design for platform games. In *AIIDE*, pages 109–111, 2006.
- [10] Gillian Smith, Jim Whitehead, Michael Mateas, Mike Treanor, Jameka March, and Mee Cha. Launchpad: A rhythm-based level generator for 2-d platformers. *IEEE Transactions on computational intelligence and AI in games*, 3(1):1–16, 2010.
- [11] Vojtěch Černý. Procedural content generation in computer games lecture at Charles University. Year: 2019/2020, <https://gamedev.cuni.cz/study/courses-history/courses-2019-2020/procedural-content-generation-in-games-201920/>.
- [12] Peter Mawhorter and Michael Mateas. Procedural level generation using occupancy-regulated extension. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 351–358. IEEE, 2010.

- [13] Noor Shaker, Julian Togelius, Georgios N Yannakakis, Ben Weber, Tomoyuki Shimizu, Tomonori Hashiyama, Nathan Sorenson, Philippe Pasquier, Peter Mawhorter, Glen Takahashi, et al. The 2010 Mario AI championship: Level generation track. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(4):332–347, 2011.
- [14] Michael O’Neil and Conor Ryan. Grammatical evolution. In *Grammatical evolution*, pages 33–47. Springer, 2003.
- [15] Noor Shaker, Miguel Nicolau, Georgios N Yannakakis, Julian Togelius, and Michael O’neill. Evolving levels for Super Mario Bros using grammatical evolution. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 304–311. IEEE, 2012.
- [16] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach, Global Edition*. Prentice Hall, 2016.
- [17] Georgios N Yannakakis and Julian Togelius. *Artificial Intelligence and Games*, volume 2. Springer, 2018.
- [18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [19] Waseem Rawat and Zenghui Wang. Deep convolutional neural networks for image classification: A comprehensive review. *Neural computation*, 29(9):2352–2449, 2017.
- [20] Wenpeng Yin, Katharina Kann, Mo Yu, and Hinrich Schütze. Comparative study of CNN and RNN for natural language processing. *arXiv preprint arXiv:1702.01923*, 2017.
- [21] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.
- [22] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dhharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.
- [23] Guillaume Lample and Devendra Singh Chaplot. Playing FPS games with deep reinforcement learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.
- [24] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*. Springer, 2003.
- [25] Sebastian Risi and Julian Togelius. Neuroevolution in games: State of the art and open challenges. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(1):25–41, 2015.

- [26] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. On-line neuroevolution applied to The open racing car simulator. In *2009 IEEE Congress on Evolutionary Computation*, pages 2622–2629. IEEE, 2009.
- [27] Erin Jonathan Hastings, Ratan K Guha, and Kenneth O Stanley. Automatic content generation in The galactic arms race video game. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(4):245–263, 2009.
- [28] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.
- [29] Jan Paredis. Coevolutionary computation. *Artificial life*, 2(4):355–375, 1995.
- [30] Sheldon M Ross. *Introduction to Probability Models*. Academic press, 2014.
- [31] Jakub Gemrot. Mario AI. <https://github.com/kefik/MarioAI>. (accessed June 25, 2020).
- [32] Vishnu Ghosh. evo-neat. <https://github.com/vishnugh/evo-NEAT>. (accessed June 8, 2020).
- [33] Julian Togelius, Sergey Karakovskiy, Jan Koutník, and Jurgen Schmidhuber. Super Mario evolution. In *2009 IEEE Symposium on Computational Intelligence and Games*, pages 156–161. IEEE, 2009.
- [34] Diego Perez, Miguel Nicolau, Michael O’Neill, and Anthony Brabazon. Evolving behaviour trees for the Mario AI competition using grammatical evolution. In *European Conference on the Applications of Evolutionary Computation*, pages 123–132. Springer, 2011.
- [35] Ahmed Khalifa, Diego Perez-Liebana, Simon M Lucas, and Julian Togelius. General video game level generation. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 253–259, 2016.
- [36] Mihaly Csikszentmihalyi and Mihaly Csikzentmihaly. *Flow: The psychology of optimal experience*, volume 1990. Harper & Row New York, 1990.
- [37] Julian Togelius and Steve Dahlskog. Patterns as objectives for level generation. In *Proceedings of the Second Workshop on Design Patterns in Games*; ACM, 2013.
- [38] Ahmed Khalifa, Fernando de Mesentier Silva, and Julian Togelius. Level design patterns in 2d games. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2019.
- [39] Gillian Smith and Jim Whitehead. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pages 1–7, 2010.
- [40] Rafael Vazquez. Gamasutra - how tough is your game? creating difficulty graphs. https://www.gamasutra.com/view/feature/134917/how_tough_is_your_game_creating_.php. (accessed July 20, 2020).

- [41] James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, October 1982.
- [42] Vojtěch Černý. Procedural generation of endless runner type of video games. Master’s thesis, Charles University, 2018.
- [43] Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O Stanley. Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions. *arXiv preprint arXiv:1901.01753*, 2019.
- [44] Alex Lubberts and Risto Miikkulainen. Co-evolving a Go-playing neural network. In *Proceedings of the GECCO-01 Workshop on Coevolution: Turning Adaptive Algorithms upon Themselves*, pages 14–19, 2001.
- [45] Chin Soon Ong, HY Quek, Kay Chen Tan, and Arthur Tay. Discovering chinese chess strategies through coevolutionary approaches. In *2007 IEEE Symposium on Computational Intelligence and Games*, pages 360–367. IEEE, 2007.
- [46] Helmut A Mayer and Peter Maier. Coevolution of neural Go players in a cultural environment. In *2005 IEEE Congress on Evolutionary Computation*, volume 2, pages 1017–1024. IEEE, 2005.
- [47] Omid David-Tabibi, H Jaap Van Den Herik, Moshe Koppel, and Nathan S Netanyahu. Simulating human grandmasters: Evolution and coevolution of evaluation functions. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1483–1490, 2009.
- [48] Siang Yew Chong, Mei K Tan, and Jonathon David White. Observing the evolution of neural networks learning to play the game of Othello. *IEEE Transactions on Evolutionary Computation*, 9(3):240–251, 2005.
- [49] JetBrains. Documenting kotlin code. <https://kotlinlang.org/docs/reference/kotlin-doc.html>. (accessed June 23, 2020).

List of Abbreviations

AI - Artificial Intelligence

ANN - Artificial Neural Network

EA - Evolutionary Algorithm

GA - Genetic Algorithm

JVM - Java Virtual Machine

MC - Markov Chain

NEAT - Neuroevolution of Augmenting Topologies

NPC - Non Player Character

PCG - Procedural Content Generation

A. Attachments

In this Appendix, we describe our attachments of this thesis.

A.1 Electronic attachment

In this section, we describe the electronic attachment. It is split into multiple directories:

- `documentation` - contains the source code documentation generated by *dokka* tool. It is a collection of HTML pages and one CSS file with entry-point being `-mario-double-evolution/index.html`. It contains descriptions of all packages, classes and chosen methods. Each package and class has its own subpage, which can be navigated to from the index page,
- `experiments` - contains experiments we did with our algorithms, which includes all four evolutions and the final coevolution. More details on structure of this folder can be found in Section A.1.1,
- `src` - contains source code of our project, whose structure is more described in Chapter 6,
- `video` - contains video which showcases one of the coevolution runs.

A.1.1 Experiments folder

In this section, we describe hierarchy of `experiments` folder in more detail. It is split into multiple subdirectories based on which algorithms were used in the given experiment:

- `ai/neuro` - contains experiments with our neuroevolution of weights algorithm. Each experiment is in its own directory which has the following form: `<name>/<configuration>`. The value of `<name>` is a name of the experiment and `<configuration>` specifies configuration of the given experiment. Its form is following:

```
<generations>:<population-size>:<fitness-function>:<mutation-probability>:<hidden-layer-size>:<input-grid-size>:<weights-range>:<dense-input>:<one-hot-encoded-enemies>.
```

For example: `50:50:D0:0.65:3:5x5:-2,2:false:false`, where the value of `<fitness-function>` is either `D0` for *distance only* fitness and `DWPx` for *distance with penalties* where `x` is the value of penalty,

- `ai/neat` - contains experiments with NEAT. Each experiment is in its own directory which has the following form: `<name>/<configuration>`. The value of `<name>` is a name of the experiment and `<configuration>` is configuration of the given experiment. Its form is following:

```
<generations>:<population-size>:<fitness-function>:<input-grid-size>:<dense-input>:<one-hot-encoded-enemies>.
```

For example: `500:100:D0:5x5:false:false`, where `<fitness-function>` acquires the same values as in the experiments with the neuroevolution of weights directory,

- `lg/pc` and `lg/pmp` - contain experiments with the evolution of Chunked Markov Chain level generator, resp. Multipass level generator. Each of the experiments uses different *fitness function* and was run 4 times. Only experiments with combined fitness function have objective function defined, which returns 1 when the win/loss ratio of the agent on a level generator is 0.5 and the farther away the ratio is from this value the lower the objective value is. In each experiment, every level generator was evaluated on 20 levels, so the maximal fitness value for *difficulty* and *linearity* is 20. When using *complexity* metric it was not normalized to $[0, 1]$ interval, and in the combined fitness the maximal value is 3 (maximally 1 for each component),
- `coev` - contains experiments with our coevolution algorithm. They are split into four subdirectories based on which two evolutionary algorithms were used in the coevolution. They are: `neuro_pc` (the neuroevolution of weights and the evolution of Chunked Markov Chain level generator), `neuro_pmp` (the neuroevolution of weights and the evolution of Multipass level generator), `neat_pc` (NEAT and the evolution of Chunked Markov Chain level generator) and `neat_pmp` (NEAT and the evolution of Multipass level generator). Because we ran each experiment 10 times, each is in its own directory. This directory then contains the following files:
 - `ai.svg` - evolution chart of all the runs of the player evolution split by black vertical lines,
 - `ai_x.ai` - serialized best evolved player from x th generation of the coevolution,
 - `coev-fitness.svg` - coevolution chart of fitness values. The fitness values of level generators is upscaled to be visible next to the fitness values of AI players,
 - `coev-objective.svg` - coevolution chart of objective values. The objective values of level generators is upscaled to be visible next to the objective values of AI players,
 - `lg.svg` - evolution chart of all the runs of the level generator evolution split by black vertical lines,
 - `lg_x.lg` - serialized best evolved level generator from the x th generation of the coevolution,
 - `*.dat files` - files representing state of the coevolution or charts' data so they can be reconstructed.

Each neuroevolution of weights and NEAT experiment was executed four times. The experiment's folder contains an evolution chart of each run, an averaged chart, and a data file for each chart so it can be reconstructed. It also contains the best evolved controller from each run, which is a serialized `MarioController` object, that can be loaded via our `ObjectStorage`.

The objective values on charts of the evolution of AI players display number of solved levels multiplied by 1000 so that it has similar scale to the fitness functions.

B. Running the evolutions

In this Appendix, we will describe how our implementation can be used to run the evolutions.

B.1 Gradle

Our project can be built using *Gradle Build Tool* of version *6.5*. Gradle requires `JAVA_HOME` environment variable to be set up properly. We recommend using Java version 11, because the Mario AI Platform had problems with some other versions.

In addition to gradle's default tasks, we have implemented some custom tasks for easier manipulation.

B.1.1 Gradle wrapper

Gradle Wrapper tool provides easier manipulation with Gradle. To get the wrapper, the user needs to download Gradle of any version at first. Then, the user needs to run the following command from the project root folder:

```
gradle wrapper --gradle-version 6.5
```

This command will download Gradle Wrapper tool for the project as well as Gradle of the given version.

If the user does not want to use the wrapper, he will need to download gradle of version *6.5* and replace `./gradlew` with `gradle` in the following commands.

B.1.2 Gradle projects

Our project is split into two Gradle projects:

- **MarioAI4J** is the *Mario AI Framework* project,
- **MarioDoubleEvolution** is the project containing our evolutions.

B.1.3 Gradle tasks - MarioAI4J

MarioAI4J contains only one notable task which needs to be run, and that is `jar`. It builds the projects and creates *jar* library which is a dependency for *MarioDoubleEvolution* project. The following command executes the task:

```
./gradlew MarioAI4J:jar
```

B.1.4 Gradle tasks - MarioDoubleEvolution

MarioDoubleEvolution project contains various custom tasks:

```
./gradlew MarioDoubleEvolution:test
```

Executes unit tests of the project.

```
./gradlew MarioDoubleEvolution:runCoevolution
```

Executes the coevolution of AI players and level generators.

```
./gradlew MarioDoubleEvolution:dokka
```

Generates *KDoc* documentation for the project.

B.2 Launchers

MarioDoubleEvolution project contains multiple entry points (*main* functions) which can be run using some IDE or the following command:

```
./gradlew runLauncher -PlaunchClass=<launcher-class>
```

<launch-class> specifies which class containing *main* function should be run. The following list contains all available launchers with their description:

- **CoEvolveExperiment** - runs coevolutions as they were run in the final experiment,
- **CoEvolveMulti** - runs one coevolution which can be configured in the source file, by modifying **NeuroEvolution** (neuroevolution of weights AI evolution), **NEATEvolution**, **PCEvolution** (Chunked Markov Chain level generator evolution) or **PMPEvolution** (Multipass level generator evolution) objects,
- **EvolveAI neat** - runs NEAT evolution of AI player once. The hyperparameters of the evolution can be specified by modifying **controllerEvolution** object in **evolve()** function. By running function **playLatest()** instead of **evolve()** from **main()**, the application will launch Super Mario game with the latest evolved agent using this launcher,
- **EvolveAI neuro** - runs the neuroevolution of weights of AI player once. The hyperparameters of the evolution can be specified by modifying object **controllerEvolution** in **evolve()** function. By running **playLatest()** function instead of **evolve()** from **main()**, the application will launch Super Mario game with the latest evolved agent using this launcher,
- **EvolveAIMany** - runs multiple evolutions of AI players (NEAT or the neuroevolution of weights), which are defined in **evolutions** array either in **doManyNeuroEvolution()** function or **doManyNEATEvolution()** function,
- **EvolveLGPC** - runs evolution of Chunked Markov Chain level generator once, which can be configured by modifying **levelGeneratorEvolution** object in **evolve()** function. By running function **playLatest()** instead of **evolve()** from **main()**, the application will launch Super Mario game with the level generator evolved in the latest run of the evolution using this launcher,

- **EvolveLGPMP** - runs evolution of Multipass level generator once, which can be configured by modifying `levelGeneratorEvolution` object in `evolve()` function. By running function `playLatest()` instead of `evolve()` from `main()`, the application will launch Super Mario game with the level generator evolved in the latest run of the evolution using this launcher,
- **EvolveLGMany** - runs multiple evolutions of a given level generator (Chunked Markov Chain or Multipass). Their parameters are defined in the `launchers` array either in function `doManyPMPEvolution()` (evolution of Multipass level generator) or `doManyPCEvolution()` (evolution of Chunked Markov Chain level generator),
- **PlayMarioAI** - starts *Super Mario* simulator using defined AI player, which will play the specified levels,
- **PlayMarioKeyboard** - starts *Super Mario* simulator with agent controlled by keyboard, which will play the specified levels.

Because Kotlin wraps functions outside of a class to a generated class of the same name as the file in which they are defined with suffix `Kt`, the actual values of the `<launch-class>` parameter also need this suffix. The parameter also expects the full name of the classes, so for better accessibility we put all the entry points to package `cz.cuni.mff.aspect.launch`. Example of a correct value of `<launch-class>` is then `cz.cuni.mff.aspect.launch.PlayMarioKeyboardKt` (for `PlayMarioKeyboard` launcher).