Experience-Driven PCG via Reinforcement Learning: A Super Mario Bros Study

Tianye Shu*†, Jialin Liu*†
* *Research Institute of Trustworthy Autonomous System*
Southern University of Science and Technology
†*Guangdong Provincial Key Laboratory of Brain-inspired Intelligent Computation*
Department of Computer Science and Engineering
Southern University of Science and Technology
Shenzhen, China
11710101@mail.sustech.edu.cn, liujl@sustech.edu.cn

Georgios N. Yannakakis
Institute of Digital Games
University of Malta
Msida, Malta
georgios.yannakakis@um.edu.mt

***Abstract*—We introduce a procedural content generation (PCG) framework at the intersections of experience-driven PCG and PCG via reinforcement learning, named ED(PCG)RL, EDRL in short. EDRL is able to teach RL designers to generate endless playable levels in an online manner while respecting particular experiences for the player as designed in the form of reward functions. The framework is tested initially in the Super Mario Bros game. In particular, the RL designers of Super Mario Bros generate and concatenate level segments while considering the diversity among the segments. The correctness of the generation is ensured by a neural net-assisted evolutionary level repairer and the playability of the whole level is determined through AI-based testing. Our agents in this EDRL implementation learn to maximise a quantification of Koster's principle of *fun* by moderating the degree of diversity across level segments. Moreover, we test their ability to design fun levels that are diverse over time and playable. Our proposed framework is capable of generating endless, playable Super Mario Bros levels with varying degrees of fun, deviation from earlier segments, and playability. EDRL can be generalised to any game that is built as a segment-based sequential process and features a built-in compressed representation of its game content.**

***Index Terms*—PCGRL, EDPCG, online level generation, procedural content generation, Super Mario Bros**

I. Introduction

Procedural content generation (PCG) [1], [2] is the algorithmic process that enables the (semi-)autonomous design of games to satisfy the needs of designers or players. As games become more complex and less linear, and uses of PCG tools become more diverse, the need for generators that are reliable, expressive, and trustworthy is increasing. Largely speaking, game content generators can produce outcomes either in an offline or in an online manner [2]. Compared to offline PCG, online PCG is flexible, dynamic and interactive but it comes with several drawbacks: it needs to be able to generate meaningful content rapidly without causing any catastrophic failure to the existing game content. Because of the many challenges that arise when PCG systems operate online (i.e. during play), only limited studies have focused on that mode of generation [3]–[6].

In this paper we introduce a framework for online PCG at the intersections of the experience-driven PCG (EDPCG) [7] and the PCG via reinforcement learning (PCGRL) [8] frameworks. The ED(PCG)RL framework, EDRL for short, enables the generation of personalised content via the RL paradigm. EDRL builds upon and extends PCGRL as it makes it possible to generate endless levels of arcade-like games beyond the General Video Game AI (GVGAI) framework [9], [10] in an online fashion. It also extends the EDPCG framework as it enables RL agents to create personalised content that is driven by experience-based reward functions. EDRL benefits from experience-driven reward functions and RL agents that are able to design in real-time based on these functions. The result is a versatile generator that can yield game content in an online fashion respecting certain functional and aesthetic properties as determined by the selected reward functions.

We test EDRL initially in *Super Mario Bros* (SMB) (Nintendo, 1985) [11] through the generative designs of RL agents that learn to optimise certain reward functions relevant to level design. In particular, we are inspired by Koster's *theory of fun* [12] and train our RL agents to moderate the level of *diversity* across SMB level segments. Moreover, we test the notion of *historical deviation* by considering earlier segment creations when diversifying the current segment. Finally, we repair defects in levels (e.g., broken pipes) via a neural net-assisted evolutionary repairer [13], and then check the playability of levels through agent-based testing. Importantly, EDRL is able to operate online in SMB as it represents the state and action via a latent vector. The key findings of the paper suggest that EDRL is possible in games like SMB; the RL agents are able

This work was supported by the National Natural Science Foundation of China (Grant No. 61906083), the Guangdong Provincial Key Laboratory (Grant No. 2020B121201001), the Program for Guangdong Introducing Innovative and Entrepreneurial Teams (Grant No. 2017ZT07X386), the Guangdong Basic and Applied Basic Research Foundation (Grant No. 2021A1515011830), the Shenzhen Science and Technology Program (Grant No. KQTD20161125143555531) and the Shenzhen Fundamental Research Program (Grant No. JCYJ20190809121403553).

to online generate playable levels of varying degrees of *fun* that deviate over time.

Beyond introducing the EDRL framework, we highlight a number of ways this paper contributes to the current state of the art. First, to the best of our knowledge, this is the first functional implementation of PCGRL in SMB, a platformer game with potentially infinite level length, that is arguably more complex than the GVGAI games studied in [8]. Second, compared to tile-scale design in [8], the proposed approach is (required to be) faster as level segments are generated online through the latent vectors of pre-trained generators. Finally, we quantify Koster's *fun* [12] as a function that maintains moderate levels of Kullback-Leibler divergence (KL-divergence) within a level teaching our RL agent to generate levels with such properties.

II. Background

A substantial body or literature has defined the area of PCG in recent years [1], [2], [7], [14]–[19]. In this section, we focus on related studies in PCG via reinforcement learning and approaches for online level generation.

A. PCG via Reinforcement Learning

Togelius *et al.* [15] have proposed three fundamental goals for PCG: "*multi-level multi-content PCG, PCG-based game design and generating complete games*". To achieve these goals, various types of PCG methods and frameworks have been researched and applied in games since then [1], [2], [7], [14]. The development of machine learning (ML) has brought revolution to PCG [2]. The combination of ML and PCG (PCGML) [16] shows great potential compared with classical PCG methods; in particular, deep learning methods have been playing an increasingly important role in PCG in recent years [19]. Furthermore, PCG methods can be used to increase the generality in ML [18]. PCGML, however, is limited by the lack of training data that is often the case in games [16]. Khalifa *et al.* [8] proposed PCG via reinforcement learning (PCGRL) which frames level generation as a game and uses an RL agent to solve it. A core advantage of PCGRL compared with existing PCGML frameworks [16] is that no training data is required. More recently, adversarial reinforcement learning has been applied to PCG [20]; specifically, a PCGRL agent for generating different environments is co-evolved with a problem solving RL agent that acts in the generated environments [20]. Engelsvoll *et al.* [21] applied a Deep Q-Network (DQN) agent to play SMB levels generated by a DQN-based level designer, which takes as input the latest columns of a played level and outputs new level columns in a tile-by-tile manner. Although, the work of [21] was the first attempt of implementing PCGRL in SMB, the emphasis in that work was not in the online generation of experience-driven PCG.

B. Online Level Generation

Content generation that occurs in real-time (i.e. online) requires rapid generation times. Towards that aim, Greuter *et al.* [22] managed to generate "pseudo infinite" virtual cities in real-time via simple constructive methods. Johnson *et al.* [23] used cellular automata to generate infinite cave levels in real-time. In [4], the model named polymorph was proposed for dynamic difficulty adjustment during the generation of levels. Stammer *et al.* [5] generated personalised and difficulty adjusted levels in the 2D platformer *Spelunky* (Mossmouth, LLC, 2008) while Shaker *et al.* formed the basis of the experience-driven PCG framework [7] by generating personalised platformer levels online [3]. Shi and Chen [24] combined rule-based and learning-based methods to generate online level segments of high quality, called constructive primitives (CPs). In the work of [6], a dynamic difficulty adjustment algorithm with Thompson Sampling [25] was proposed to combine these CPs.

III. EDRL: Learn to Design Experiences via RL

Our aim is to introduce a framework that is able to generate endless, novel yet consistent and playable levels, ideally for any type of game. Given an appropriate level-segment generator, our level generator selects iteratively the suitable segment to be concatenated successively to the current segment to build a level. This process resembles a jigsaw puzzle game, of which the goal is to pick suitable puzzle pieces (i.e., level segment) and put them together to match a certain pattern or "style".

Playing this jigsaw puzzle can be modelled as a Markov Decision Process (MDP) [26]. In this MDP, a state s models the current puzzle piece (level segment) and an action is the next puzzle piece to be placed. The reward $r(s, a)$ evaluates how these two segments fit. The next state s' after selecting a is set as a itself assuming deterministic actions, i.e., $s' = MDP(s, a) = a$. An optimal action a^* at state s is defined as the segment that maximises the reward $r(s, a)$ if being placed after segment s, i.e., $a^* = \arg\max_{a \in \mathcal{A}} r(s, a)$. $\mathcal{A}$ denotes the actions space, i.e., the set of all possible segments. An RL agent is trained to learn the optimal policy π^* that selects the optimal segments, thus $a^* = \pi^*(s)$. Endless-level generation can be achieved if this jigsaw puzzle game is played infinitely.

One could argue that the jigsaw puzzle described above does not satisfy the Markov property as the level consistency and diversity should be measured based on the current and the concatenated segment. When a human plays such a game, however, she can only perceive the current game screen. In fast-paced reactive games, the player's short term memory is highly active during reactive play and, thus episodic memory of the game's surroundings is limited to a local area around play [27]. Therefore, we can assume that the Markov property is satisfied to a good degree within a suitable length of such level segments.

The general overview of the EDRL framework is shown in Fig. 1. The framework builds on EDPCG [7] and PCGRL [8] and extends them both by enabling experience-driven PCG via the RL paradigm. According to EDRL an RL agent learns to design content with certain player experience and aesthetic aspects (*experience model*) by interacting with the RL environment which is defined through a *content representation*.

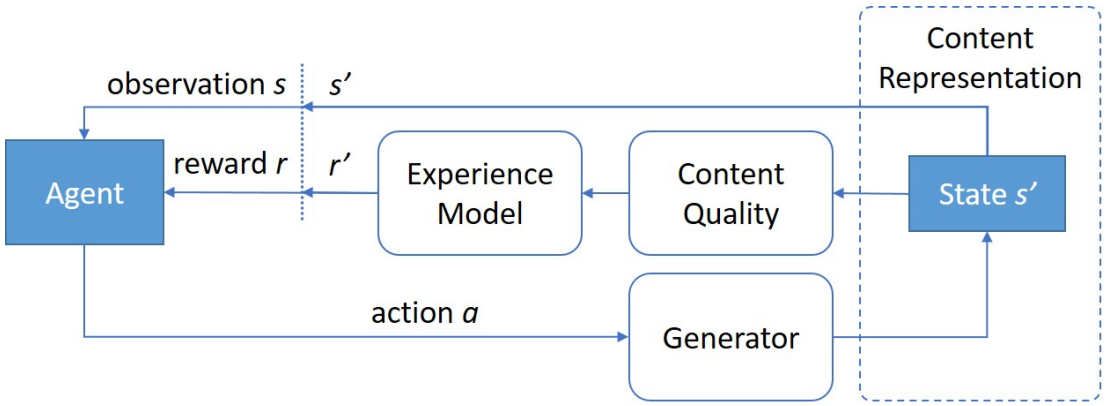


Fig. 1: General overview of the EDRL framework interweaving elements of both the EDPCG and the PCGRL frameworks. White rounded boxes and blue boxes depict components of EDPCG [7] and PCGRL [8], respectively. Content representation (i.e., environment in RL terms) is a common component across the two frameworks and is depicted with a dashed line.

The *content quality* component guarantees that the experience model will consider content of certain quality (e.g., tested via gameplay simulations). The RL designer takes an action that corresponds to a generative act that alters the state of the represented content (s') and receives a reward r' through the *experience model* function. The agent iteratively traverses the design (state-action representation) and experience (reward) space to find a design policy that optimises the *experience model*.

The framework is directly applicable to any game featuring levels that can be segmented and represented rapidly through a compressed representation such as a latent vector. This includes Atari-like 2D games [9], [28] but can also include more complex 3D games (e.g., VizDoom [29]) if latent vectors are available and can synthesise game content.

A. Novelty Of EDRL

The presented EDRL framework sits at the intersection of experience-driven PCG and PCGRL being able to create personalised (i.e., experience-tailored) levels via RL agents in a real-time manner. EDRL builds on the core principles of PCGRL [8] but it extends it in a number of ways. First, vanilla PCGRL focuses on training an RL agent to design levels from scratch. Our framework, instead, teaches the agent to learn to select suitable level segments based on content and game-play features. Another key difference is the action space of RL agents. Instead of tile-scale design, our designer agent selects actions in the latent space of the generator and uses the output segment to design the level in an online manner. The length of the level is not predefined in our framework. Therefore, game levels can in principle be generated and played endlessly.

For designing levels in real-time, we consider the diversity of new segments compared to the ones created already, which yields reward functions for fun and deviation over time; earlier work (e.g., [4], [6]) largely focused on objective measures for dynamic difficulty adjustment. Additionally, we use a repairer that corrects level segments without human intervention, and game-playing agents that ensure their playability. It is a hard challenge to determine which level segment will contribute best to the level generation process—i.e. a type of credit assignment problem for level generation. To tackle this challenge, Shi and Chen [6] formulated dynamic difficulty adjustment as an MDP [26] with binary reward and a Thompson sampling method. We, instead, frame online level generation as an RL game bounded by any reward function which is not limited to difficulty, but rather to player experience.

IV. EDRL for Mario: MarioPuzzle

In this section we introduce an implementation of EDRL for SMB level generation and the specific reward functions designed and used in our experiments (Section IV-D). EDRL in this implementation enables *online* and *endless* generation of content under functional (i.e., playbility) and aesthetic (e.g., fun) metrics. As illustrated in Fig. 2, the implementation for SMB—namely *MarioPuzzle*[1]—features three main components: (i) a generator and repairer of non-defective segments, (ii) an artificial SMB player that tests the playability of the segments and (iii) an RL agent that plays this *MarioPuzzle* endless-platform generation game. We describe each component in dedicated sections below.

A. Generate and Repair via the Latent Vector

As presented earlier in Section III, both the actions and states in this MDP represent different level segments. Our framework naturally requires a level segment generator to operate. For that purpose we use and combine the *MarioGAN* [30] generator and *CNet-assisted Evolutionary Repairer* [13] to, respectively, generate and repair the generated level segments. The CNet-assisted Evolutionary Repairer has shown to be capable of determining wrong tiles in segments generated by MarioGAN [30] and repairing them [13]; hence,

[1]Available on GitHub: https://github.com/SliverySky/mariopuzzle

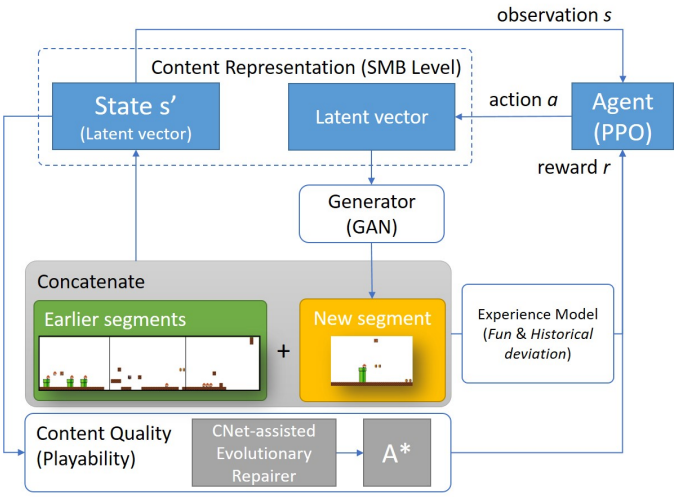


Fig. 2: Implementing EDRL on Super Mario Bros: the *MarioPuzzle* framework.

the *correctness* of generated segments is guaranteed. In particular, we train a GAN[2] on fifteen SMB levels of three types (overworld, underground and athletic) in VGLC [31]. The CNet-assisted Evolutionary Repairer trained by [13] is used directly and unmodified [3].

It is important to note that we are not using a direct tile-based representation as in [8] or one-hot encoded tiles as in [21]; instead we use the latent vector of *MarioGAN* to represent the agent's actions and states for the RL agent. Thus, the selected action a or state s are sampled from the latent space, rather than the game space.

B. The AI Player

The A^* agent in the Mario AI framework [32] is used as an artificial player for determining the playability of generated segments[4].

C. The RL Designer

MarioPuzzle is embedded into the OpenAI gym [33] and the PPO algorithm [34] is used to train our agent [35]. The training procedure of *MarioPuzzle* is described in Algorithm 1. The actions and states are represented by latent vectors of length 32. When training the PPO agent, the initial state is a playable segment randomly sampled from the latent space of the trained segment generator, as shown in Algorithm 1. The latent vector of the current level segment feeds the current observation of the PPO agent. Then, an action (i.e., a latent vector) is selected by the PPO agent and used as an input of the generator to generate a new segment. The repairer determines and fixes broken pipes in the new segment. The fixed segment is concatenated to the earlier one and the A^* agent tests if the addition of the new segment is playable. Then, the latent vector of this new segment (same as action) is returned as a new observation to the agent. The agent receives an immediate reward for taking an action in a particular state. The various reward functions we considered in this study are based on both content and game-play features and are detailed below.

[2]https://github.com/schrum2/GameGAN
[3]https://github.com/SUSTechGameAI/MarioLevelRepairer
[4]https://github.com/amidos2006/Mario-AI-Framework

Algorithm 1 Training procedure of *MarioPuzzle*.

Require: $\mathcal{G} : [-1, 1]^{32} \mapsto segment$, trained GAN
Require: $\mathcal{F} : segment \mapsto segment$, trained repairer
Require: $\mathcal{P} : segment \mapsto [0, 1]$, A^* agent
Require: $\pi : [-1, 1]^{32} \mapsto [-1, 1]^{32}$, RL agent
Require: $Reward$, reward function
Require: T: maximum number of training epochs
Require: N: maximum segment number of one game

$t \leftarrow 0$
while $t < T$ **do**
  $PreS \leftarrow Empty\ List$
  $n \leftarrow 0$
  $s_n \leftarrow NULL$
  repeat
    $s \leftarrow$ uniformly sampled from $[-1, 1]^{32}$
    $S \leftarrow \mathcal{G}(s)$ // *Generate a segment*
    $S \leftarrow \mathcal{F}(S)$ // *Repair the segment if applicable*
    $isPlayable \leftarrow \mathcal{P}(S)$ // *Play this segment*
    if $isPlayable$ **then**
      $s_n \leftarrow v$
    end if
  until s_n is not $NULL$
  Add S to $PreS$
  while $isPlayable$ and $n < N$ **do**
    $a_n \leftarrow \pi(s_n)$ // *Select an action*
    $S \leftarrow \mathcal{G}(a_n)$ // *Generate a segment*
    $S \leftarrow \mathcal{F}(S)$ // *Repair the segment if applicable*
    $isPlayable \leftarrow \mathcal{P}(S)$ // *Play this segment*
    $score_{s_n,a_n} \leftarrow Reward(S)$ with previous $PreS$ // *According to metrics*
    Update π with $score_{s_n,a_n}$
    Update $PreS$ with S // *According to metrics*
    $n = n + 1$
  end while
  $t = t + 1$
end while
return π

D. Reward Functions

Designing a suitable *reward* function for the RL agent to generate desired levels is crucial. In this implementation of EDRL, we design three metrics that formulate various reward functions ($Reward$ in Algorithm 1) aiming at guiding the RL agent to learn to generate playable levels with desired player experiences.

1) Moderating Diversity Makes Fun!: Koster's *theory of fun* [12] suggests that a game is fun when the patterns a player perceives are neither too unfamiliar (i.e., changeling) nor too familiar (i.e., boring). Inspired by this principle, when our agent concatenates two segments, we assume it should

keep the diversity between them at moderate levels; too high diversity leads to odd connections (e.g., mix of styles) whereas too low diversity yields segments that look the same. To do so, we first define a diversity measure, and then we moderate diversity as determined from human-designed levels.

When a player plays through a level, the upcoming level segment is compared with previous ones for diversity. We thus define the diversity of a segment through its dissimilarity to previous segments. While there have been several studies focusing on quantifying content similarity (e.g., [36]), we adopt the tile-based KL-divergence [37] as a simple and efficient measure of similarity between segments. More specifically diversity, D, is formulated as follows:

$$D(S) = \frac{1}{n+1} \sum_{i=0}^{n} KL(S, S_{W_i}), \tag{1}$$

where S is a generated level segment with height h and width w; $KL(a, b)$ is the tile-based KL-divergence that considers the standard KL-Divergence between the distributions over occurrences of tile patterns in two given level segments, a and b [37]. We define a window W with the same size as S. S_W represents the segment contained in the window W. A sliding window moves from the position of S ($S_{W_0} = S$) to the previous segment n times with stride d. The parameter n limits the number of times that the window moves. According to (1), larger n values consider more level segments in the past. After preliminary hyper-parameter tuning, we use a 2×2 window and $\epsilon = 0.001$ for calculating tile-based KL-divergence [37]. n and d are set as 3 and 7 in this work.

Once we have diversity defined, we attempt to moderate diversity by considering the fifteen human-authored SMB levels used for training our GAN. We thus calculate the average and standard deviation of D for all segments across the three different level types of our dataset. Unsurprisingly, Table I shows that different level types yield significantly different degrees of diversity. It thus appears that by varying the degree of diversity we could potentially vary the design style of our RL agent. Randomly concatenating level segments, however, cannot guarantee moderated diversity as its value depends highly on the expressive range of the generator. On that end, we moderate the diversity of each segment in predetermined ranges—thereby defining our *fun* (F) reward function—as follows.

$$F(S) = \begin{cases} -(D(S) - u)^2, & \text{if } D(S) > u \\ -(D(S) - l)^2, & \text{if } D(S) < l \\ 0, & \text{otherwise,} \end{cases} \tag{2}$$

where l and u denote the lower and upper bounds of diversity, respectively. According to the analysis on human-authored SMB levels (Table I), we assume that the diversity of *overworld* levels is moderate and arbitrarily set $l = 0.60 - 0.34 = 0.26$ and $u = 0.60 + 0.34 = 0.94$, based on the diversity values obtained for this type of levels.

TABLE I: Average diversity values and corresponding standard deviations of segments across the three types of SMB levels.

Type	#Level	$\sum$#Segment	D [Eq. (1)]
Overworld	9	1,784	0.60 ± 0.34
Athletic	4	534	0.32 ± 0.25
Underground	2	305	1.11 ± 0.59
Total	15	2,623	0.60 ± 0.42

2) Historical Deviation: Inspired by the notion of novelty score [38], we consider *historical deviation*, H, as a measure that encourages our agent to generate segments that deviate from earlier creations. In particular, we define H of a segment as the average similarity of the k most similar segments among the $m > k$ previous segments, as formalised in (3).

$$H(S) = \frac{1}{k} \sum_{j=1}^{k} KL(S, S_{I_j}), \tag{3}$$

where I_j is the index of the j^{th} most similar segment among previous m segments compared to S; m represents the number of segments that our RL agent holds in its memory. This parameter is conceptually similar to the *sparseness* parameter in novelty search [39]. After preliminary hyper-parameter tuning, m and k are set as 20 and 10, respectively, in the experiments of this paper.

3) Playablility: The playablility of a newly generated segment is tested by an A^* agent. The generated segment, however, is not played on its own; its concatenation to the three previous segments is played as a whole instead, to ensure that the concatenation will not yield unplayable levels. When testing, Mario starts playing from its ending position in the previous level segment. A segment S is determined as playable only if Mario succeeds in reaching the right-most end of S.

Playability is essential in online level generation. Therefore, when an unplayable segment is generated, the current episode of game design ends—see Algorithm 1—and the reward returned, $P(S)$, is set as the number of segments that the A^* Mario managed to complete. With such a reward function the agent is encouraged to generate the longest playable levels possible.

V. Design Experiments Across Reward Functions

In this section, we test the effectiveness of the metrics detailed in Section IV-D through empirical experimentation. We use each of the fun and historical deviation metrics as independent reward functions and observe their impact on generated segments. Then, we use combinations of these metrics and playability to form new reward functions.

A. Experimental Details and Results

When calculating F, the size of the sliding window and the segment are both 14×14 and the control parameters n and d of Eq. (1) are set as 3 and 7, respectively. When calculating H, the control parameters m and k of Eq. (3) are set as 20 and

TABLE II: Evaluation metrics of levels generated across different RL agents over 300 levels each. The Table presents average values and corresponding standard deviations across the 300 levels. π_* refers to the trained policy using the $*$ reward function. π_R refers to a random agent that designs SMB levels. $\overline{F}$, $\overline{H}$ and $\overline{P}$ are respectively the F, H and P values averaged over playable segments by the A^* agent; in addition to $\overline{F}$, the $\overline{F}_b$ value appearing in square brackets refers to the average percentage of playable segments within the bounds of moderate diversity (cf. Section IV-D1). In addition to the three evaluation metrics, the Table presents the number of gaps, pipes, enemies, bullet blocks, coins, and question-mark blocks that exist in the generated playable segments. Values in bold indicate the highest value in the column.

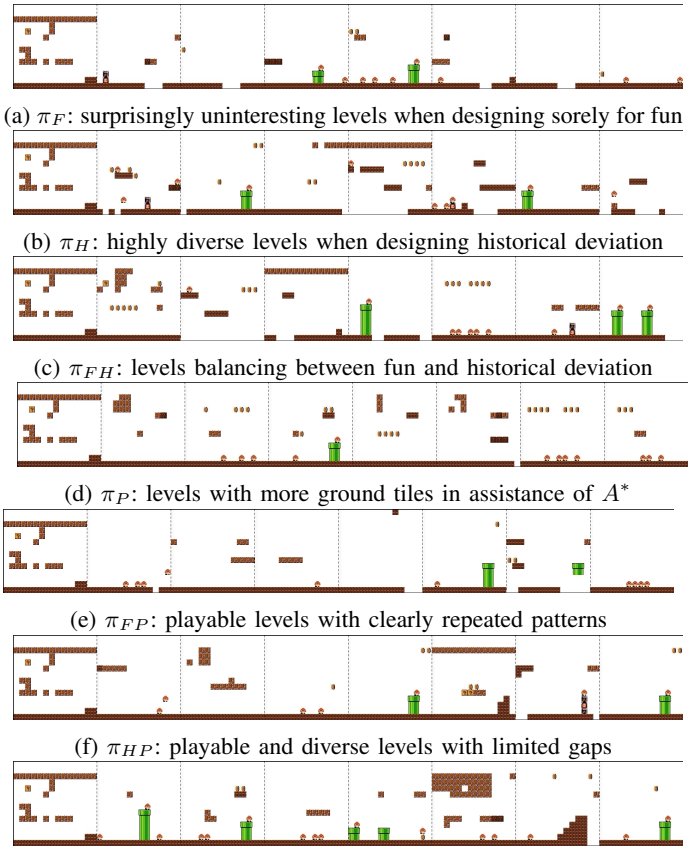
Agent	Evaluation metrics			Number of level elements in generated segments					
	$\overline{F}$ [$\overline{F}_b$]	$\overline{H}$	$\overline{P}$	Gaps	Pipes	Enemies	Bullets	Coins	Question-marks
π_F	**-0.005±0.044 [87.1±14.1]**	0.86±0.28	29.6±28.3	0.60±0.40	0.43±0.17	**2.11±0.63**	0.05±0.13	0.64±0.52	**1.00±0.59**
π_H	-0.092±0.092 [57.0±18.8]	**1.43±0.32**	24.2±21.8	0.73±0.34	0.40±0.31	1.48±0.58	0.09±0.10	1.10±0.62	0.68±0.55
π_{FH}	-0.065±0.086 [63.4±21.8]	1.38±0.38	16.4±16.2	0.74±0.37	**0.49±0.33**	1.69±0.74	0.11±0.19	1.06±0.79	0.53±0.59
π_P	-0.023±0.013 [76.7±5.1]	0.72±0.07	97.3±11.8	0.12±0.04	0.15±0.04	1.58±0.15	0.10±0.03	2.37±0.25	0.29±0.09
π_{FP}	-0.032±0.017 [75.2±5.2]	0.83±0.09	96.6±14.4	0.18±0.05	0.17±0.04	1.28±0.17	0.10±0.04	**2.51±0.25**	0.46±0.11
π_{HP}	-0.037±0.020 [74.2±5.7]	0.84±0.09	97.0±13.9	0.18±0.07	0.22±0.05	1.40±0.20	0.10±0.04	2.48±0.29	0.46±0.18
π_{FHP}	-0.034±0.018 [74.0±6.0]	0.84±0.09	**97.4±13.7**	0.23±0.17	0.18±0.09	1.19±0.40	**0.12±0.04**	2.43±0.36	0.55±0.13
π_R	-0.064±0.098 [64.3±21.9]	1.35±0.37	16.0±14.3	**0.82±0.44**	0.45±0.36	1.53±0.75	0.10±0.18	0.94±0.71	0.52±0.55

10, respectively. For each RL agent presented in this paper, a PPO algorithm is used and trained for 10^6 epochs.

In all results and illustrations presented in this paper, F, H and P refer to independent metrics that consider, respectively, the fun (2), historical deviation (3) and playability (cf. Section IV-D3) of a level. The notation π_F refers to the PPO agent trained solely through the F reward function. Similarly, π_{FH} refers to the agent trained with the sum of F and H as a reward, while π_{FHP} refers to the one trained with the sum of F, H and P as a reward. When a reward function is composed by multiple metrics, each of the metrics included is normalised within $[0, 1]$ based on the range determined by the maximum and minimum of its $1,000$ most recent values.

For testing the trained agents, 30 different level segments are used as initial states. Given an initial state, an agent is tested independently 10 times for a maximum of 100 segments—or until an unplayable segment is reached if P is a component of the reward function. As a result, each agent designs 300 levels of potentially different number of segments when P is considered; otherwise, the generation of an unplayable segment will not terminate the design process, thus 100 segments will be generated. For comparison, a random agent, referred to as π_R, is used to generate levels by randomly sampling up to 100 segments from the latent space or till an unplayable one is generated.

The degrees of fun, historical deviation, and playability of these generated levels are evaluated and summarised in Table II. Table II also showcases average numbers of core level elements in generated levels, including pipes, enemies and question-mark blocks. Figure 3 illustrates a number of arbitrarily chosen segments clipped from levels generated by each RL agent.



(a) π_F: surprisingly uninteresting levels when designing sorely for fun

(b) π_H: highly diverse levels when designing historical deviation

(c) π_{FH}: levels balancing between fun and historical deviation

(d) π_P: levels with more ground tiles in assistance of A^*

(e) π_{FP}: playable levels with clearly repeated patterns

(f) π_{HP}: playable and diverse levels with limited gaps

(g) π_{FHP}: fun, diverse and playable levels

Fig. 3: Example segments clipped from levels generated by RL agents trained with different reward functions. The key characteristics of each EDRL policy are outlined in the corresponding captions.

B. Analysis of Findings

The key findings presented in Table II suggest that the evaluation functions of $\overline{F}$ and $\overline{H}$ across all generated levels reach their highest value when the corresponding reward functions are used. In particular, the π_F policy yields appropriate levels of diversity in 87.1 out of 100 segments, on average; higher than any other policy (see $\overline{F}_b$ values in Table II). Similarly, $\overline{H}$

reaches its highest value on average (1.43) when policy π_H is used. Finally, playabilty is boosted when P is used as a reward function but it reaches its peak value of 97.4 out of 100 segments on average when it is combined with both F and H.

π_F focuses primarily on enemy and question-mark block placement, but the levels it generates are far from looking diverse (see Fig. 3a). π_H does not yield any distinct level element and results in rather diverse yet unplayable levels (see Fig. 3b). The combination of F and H without playability (π_{FH}) augments the number of pipes existent in the level and offers interesting but largely unplayable levels (see Table II and Fig. 3c).

When P is considered by the reward function, the outcome is a series of playable levels (i.e., the average P value is over 96 out of 100 segments per level). The most interesting agent from all the combinations of P with the other two reward functions, however, seems to be the policy that is trained on all three (π_{FHP}; see Fig. 3g). That policy yields highly playable levels that maintain high levels of fun—i.e. on average, 74 out of 100 segments reach appropriate levels of diversity—and historical deviation (0.84 on average). The π_{FHP} agent appears to design more playable levels, but at the same time, the generated levels yield interesting patterns and have more gaps, coins and bullet blocks that increase the challenge and augment the exploration paths available for the player (see Table II and Fig. 3g). Based on the performance of π_{FHP}—compared against all other SMB level design agents explored—the next section puts this agent under the magnifying glass and analyses its online generation capacities.

VI. Online Endless-level Generation

To test the performance of the trained π_{FHP} agent for online level generation, we let it generate 300 levels (10 trials from 30 initial segments) composed of 100 playable segments each; although, in principle, the agent can generate endless levels online. The results obtained are shown in Table III and are discussed in this section. As a baseline we compare the performance of π_{FHP} against π_{FH}, i.e., an agent that does not consider playability during training.

When generating a level, it is necessary to repair the defects and test its playability. The faulty tiles of the generated segments before and after repairing are detected by CNet [13] (see column 6 in Table III). Clearly, the levels generated by policy π_{FH} feature more faulty tiles than levels generated by π_{FHP}. After visualising the generated levels, we observe that the pipes in the levels generated by π_{FH} have diverse locations. Most of the pipes in levels generated by π_{FHP}, instead, are located in similar positions (i.e., near the middle bottom of the segment). We assume that the segments with this pattern are more likely to be chosen by the π_{FHP} agent as they are easier to complete by the testing agent.

The repairer operation by the CNet-assisted Evolutionary Algorithm only satisfies the logical constraints but does not guarantee the playability of the level. If a generated segment is unplayable, the RL agent will re-sample a new segment with one of the two methods: either by (i) sampling a new action according to its policy (ii) sampling a new action randomly from a normal distribution, and then clipping it into $[-1, 1]^{32}$. This resampling process repeats until a playable segment is obtained or a sampling counter reaches 20. Note that this resampling technique is only used in this online level generation test to ensure the generation of levels composed by playable segments only. Resampling is not enabled during training as an RL agent trained with playability as part of its reward is expected to learn to generate playable segments.

To evaluate the real-time generation efficiency of *MarioPuzzle* we record the number of times resampling is required (see column 3 in Table III), the total number of resamplings (see column 4 in Table III), and the time taken for generating a level of 100 playable segments (see column 5 in Table III). Unsurprisingly, the π_{FH} agent—which is trained without playability as part of its reward—appears to generate more unplayable segments and is required to resample more.

As a baseline of real-time performance we asked three human players (students in our research group) to play 8 levels that are randomly selected from the level creations of π_{FHP}. Their average playing time for one segment was $2.7s$, $3.1s$ and $7.6s$. It thus appears that the average segment generation time (see column 5 in Table III) of π_{FHP} is acceptable for online generation when compared to the time needed for a human player to complete one segment.

According to Table III, π_{FHP} with random resampling never fails in generating playable 100-segment long levels. Comparing π_{FHP} with π_{FH}, it is obvious that integrating playability into the reward function can reduce the probability of generating unplayable segments and resampling times, and, in turn, make the online generation of playable levels easier and faster.

Figure 4 displays examples from the levels generated by π_{FHP}. The EDRL designer resolves the playability issues by placing more ground tiles while maintaining appropriate levels of F and H. It is important to remember that EDRL in this work operates and designs fun, diverse and playable levels for an A^* playing agent. The outcome of Fig. 4 reflects on the attempt of the algorithm to maximise all three reward functions for this particular and rather predictable player, thereby offering largely linear levels without dead-ends, with limited gaps (ensuring playability) and limited degrees of exploration for the player. The resulting levels for agents that depict more varied gameplay are expected to vary substantially.

VII. Discussion and Future Work

In this paper we introduced EDRL as a framework, and instantiated it in SMB as *MarioPuzzle*. We observed the capacity of *MarioPuzzle* to generate endless SMB levels that are playable and attempt to maximise certain experience metrics. As this is the first instance of EDRL in a game, there is a number of limitations that need to be explored in future studies; we discuss these limitations in this section.

By integrating playability in the reward function we aimed to generate playable levels that are endless in principle [40].

TABLE III: Generating 100-segment long levels. All values—expect from the number of failed generations—are averaged across 300 levels (10 trails each from 30 different initial level segments). In the "Generation Time" column, "Sample" refers to the time averaged over the total number of generated segments, including unplayable and playable ones, while "Segment" refers to the time averaged over successful level generations.

Column index	1	2	3	4		5		6	
Agent	Resampling method	Failed generations	Unplayable segments	Resamples Max	Resamples Total	Generation Time (s) Segment	Generation Time (s) Sample	Faulty tiles Original	Faulty tiles Repaired
π_{FH}	Random	205/300	6.15	1.97	7.24	1.09	1.02	50.2	8.4
	Policy	196/300	6.36	2.18	7.61	1.11	1.03	54.1	10.2
π_{FHP}	Random	0/300	0.10	0.11	0.11	0.79	0.79	6.1	0.3
	Policy	5/300	0.12	0.11	0.12	0.79	0.79	6.1	0.3

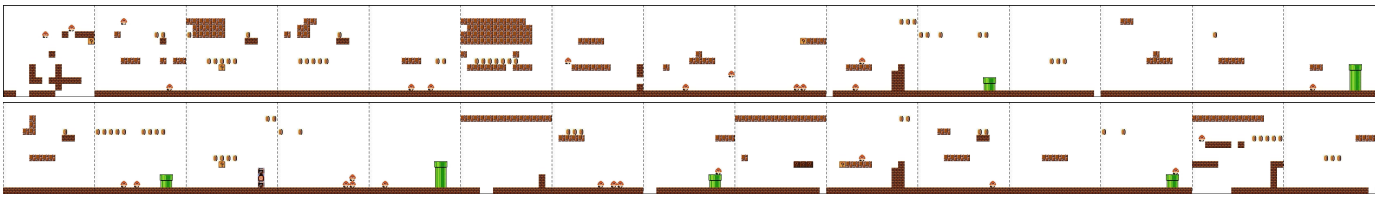


Fig. 4: Example of partial game levels generated online by the π_{FHP} agent.

As a result, the RL agent generates more ground tiles to make it easier for the A^* to pass though. The generated levels highly depend both on the reward function and the behaviour of the test agent. Various human-like agents and reward functions will need to be studied in future work to encourage the RL agent to learn to select suitable level segments for different types of players. Moreover, the behaviour of human-like agents when playing earlier segments can be used to continuously train the RL agent. In that way our RL designer may continually (online) learn through the behaviour of a human-like agent and adapt to the player's skill, preferences and even annotated experiences.

In this initial implementation of EDRL, we used a number of metrics to directly represent player experience in a theory-driven manner [2]. In particular, we tested expressions of fun and historical deviation for generating segments of game levels. As future work, we intend to generate levels with adaptive levels of diversity, surprise [41], [42], and/or novelty and investigate additional proxies of player experience including difficulty-related functions stemming from the theory of flow [43]. Similarly to [44], [45], we are also interested in applying multi-objective optimisation procedures to consider simultaneously the quality and diversity of levels, instead of their linear aggregation with fixed weights. Viewing the EDRL framework from an intrinsic motivation [46] or an artificial curiosity [47] perspective is another research direction we consider. All above reward functions, current and future ones, will need to be cross-verified and tested against human players as in [48].

While EDRL builds on two general frameworks and it is expected to operate in games with dissimilar characteristics, our plan is to test the degrees to which EDRL is viable and scalable to more complex games that feature large game and action space representations. We argue that given a learned game content representation—such as latent vector of a GAN or an autoencoder—and a function that is able to segment the level, EDRL is directly applicable. Level design was our test-bed in this study; as both EDPCG and PCGRL (to a lesser degree) have explored their application to other forms of content beyond levels, EDRL also needs to be tested to other types of content independently or even in an orchestrated manner [49].

VIII. Conclusion

In this paper, we introduced a novel framework that realises personalised online content generation by interweaving the EDPCG [7] and the PCGRL [8] frameworks. We test the ED(PRCG)RL framework, EDRL in short, in Super Mario Bros and train RL agents to design endless and playable levels that maximise notions of *fun* [12] and historical deviation. To realise endless generation in real-time, we employ a pre-trained GAN generator that designs level segments; the RL agent selects suitable segments (as represented by their latent vectors) to be concatenated to the existing level. The application of EDRL in SMB makes online level generation possible while ensuring certain degree of fun and deviation across level segments. The generated segments are automatically repaired by a CNet-assisted Evolutionary Algorithm [13] and tested by an A^* agent that guarantees playability. This initial study showcases the potential of EDRL in fast-paced games like SMB and opens new research horizons for realising experience-driven PCG though the RL paradigm.

References

[1] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural content generation in games*. Springer, 2016.

[2] G. N. Yannakakis and J. Togelius, *Artificial intelligence and games*. Springer, 2018, vol. 2.

[3] N. Shaker, G. Yannakakis, and J. Togelius, "Towards automatic personalized content generation for platform games," in *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2010, pp. 63–68.

[4] M. Jennings-Teats, G. Smith, and N. Wardrip-Fruin, "Polymorph: A model for dynamic level generation," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 5, no. 1, 2010.

[5] D. Stammer, T. Günther, and M. Preuss, "Player-adaptive spelunky level generation," in *2015 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2015, pp. 130–137.

[6] P. Shi and K. Chen, "Learning constructive primitives for real-time dynamic difficulty adjustment in Super Mario Bros," *IEEE Transactions on Games*, vol. 10, no. 2, pp. 155–169, 2017.

[7] G. N. Yannakakis and J. Togelius, "Experience-driven procedural content generation," *IEEE Transactions on Affective Computing*, vol. 2, no. 3, pp. 147–161, 2011.

[8] A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, "PCGRL: Procedural content generation via reinforcement learning," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 16, no. 1, 2020, pp. 95–101.

[9] D. Perez-Liebana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius, and S. M. Lucas, "General video game AI: A multitrack framework for evaluating agents, games, and content generation algorithms," *IEEE Transactions on Games*, vol. 11, no. 3, pp. 195–214, 2019.

[10] D. Perez-Liebana, S. M. Lucas, R. D. Gaina, J. Togelius, A. Khalifa, and J. Liu, *General Video Game Artificial Intelligence*. Morgan & Claypool Publishers, 2019, https://gaigresearch.github.io/gvgaibook/.

[11] R. Nintendo, "Super Mario Bros," *Game [NES].(13 September 1985). Nintendo, Kyoto, Japan*, 1985.

[12] R. Koster, *Theory of fun for game design*. "O'Reilly Media, Inc.", 2013.

[13] T. Shu, Z. Wang, J. Liu, and X. Yao, "A novel cnet-assisted evolutionary level repairer and its applications to Super Mario Bros," in *2020 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2020.

[14] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[15] J. Togelius, A. J. Champandard, P. L. Lanzi, M. Mateas, A. Paiva, M. Preuss, and K. O. Stanley, "Procedural content generation: Goals, challenges and actionable steps." Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

[16] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius, "Procedural content generation via machine learning (PCGML)," *IEEE Transactions on Games*, vol. 10, no. 3, pp. 257–270, 2018.

[17] B. De Kegel and M. Haahr, "Procedural puzzle generation: A survey," *IEEE Transactions on Games*, vol. 12, no. 1, pp. 21–40, 2020.

[18] S. Risi and J. Togelius, "Increasing generality in machine learning through procedural content generation," *Nature Machine Intelligence*, vol. 2, no. 8, pp. 428–436, 2020.

[19] J. Liu, A. Khalifa, Snodgrass, S. Risi, G. N. Yannakakis, and J. T. Togelius, "Deep learning for procedural content generation," *Neural Computing and Applications volume*, vol. 33, pp. 19–37, 2021.

[20] L. Gisslén, A. Eakins, C. Gordillo, J. Bergdahl, and K. Tollmar, "Adversarial reinforcement learning for procedural content generation," in *2021 IEEE Conference on Games (CoG)*. IEEE, 2021, p. accepted.

[21] R. N. Engelsvoll, A. Gammelsrød, and B.-I. S. Thoresen, "Generating levels and playing Super Mario Bros. with deep reinforcement learning using various techniques for level generation and deep q-networks for playing," Master's thesis, University of Agder, 2020.

[22] S. Greuter, J. Parker, N. Stewart, and G. Leach, "Real-time procedural generation of pseudo infinite cities," in *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, 2003, pp. 87–ff.

[23] L. Johnson, G. N. Yannakakis, and J. Togelius, "Cellular automata for real-time generation of infinite cave levels," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 2010, pp. 1–4.

[24] P. Shi and K. Chen, "Online level generation in Super Mario Bros via learning constructive primitives," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2016, pp. 1–8.

[25] W. R. Thompson, "On the likelihood that one unknown probability exceeds another in view of the evidence of two samples," *Biometrika*, vol. 25, no. 3/4, pp. 285–294, 1933.

[26] R. Bellman, "A markovian decision process," *Journal of mathematics and mechanics*, vol. 6, no. 5, pp. 679–684, 1957.

[27] E. Tulving, "Episodic memory: From mind to brain," *Annual review of psychology*, vol. 53, no. 1, pp. 1–25, 2002.

[28] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.

[29] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaśkowski, "Vizdoom: A Doom-based AI research platform for visual reinforcement learning," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2016, pp. 1–8.

[30] V. Volz, J. Schrum, J. Liu, S. M. Lucas, A. Smith, and S. Risi, "Evolving mario levels in the latent space of a deep convolutional generative adversarial network," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2018, pp. 221–228.

[31] A. J. Summerville, S. Snodgrass, M. Mateas, and S. Ontanón, "The VGLC: The video game level corpus," in *Proceedings of 1st International Joint Conference of DiGRA and FDG*, 2016. [Online]. Available: https://arxiv.org/abs/1606.07487

[32] N. Shaker, J. Togelius, G. N. Yannakakis, B. Weber, T. Shimizu, T. Hashiyama, N. Sorenson, P. Pasquier, P. Mawhorter, G. Takahashi *et al.*, "The 2010 Mario AI championship: Level generation track," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 4, pp. 332–347, 2011.

[33] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," *arXiv preprint arXiv:1606.01540*, 2016.

[34] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[35] I. Kostrikov, "Pytorch implementations of reinforcement learning algorithms," https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail, 2018.

[36] A. Isaksen, C. Holmgård, and J. Togelius, "Semantic hashing for video game levels," *Game Puzzle Des*, vol. 3, no. 1, pp. 10–16, 2017.

[37] S. M. Lucas and V. Volz, "Tile pattern KL-divergence for analysing and evolving game levels," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2019, pp. 170–178.

[38] J. Lehman and K. O. Stanley, "Exploiting open-endedness to solve problems through the search for novelty." in *ALIFE*. Citeseer, 2008, pp. 329–336.

[39] ——, "Novelty search and the problem with objectives," in *Genetic programming theory and practice IX*. Springer, 2011, pp. 37–56.

[40] J. Togelius, S. Karakovskiy, and R. Baumgarten, "The 2009 Mario AI competition," in *IEEE Congress on Evolutionary Computation*. IEEE, 2010, pp. 1–8.

[41] G. N. Yannakakis and A. Liapis, "Searching for surprise." ICCC, 2016.

[42] D. Gravina, A. Liapis, and G. N. Yannakakis, "Quality diversity through surprise," *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 4, pp. 603–616, 2018.

[43] M. Csikszentmihalyi, S. Abuhamdeh, and J. Nakamura, "Flow," in *Flow and the foundations of positive psychology*. Springer, 2014, pp. 227–238.

[44] G. Cideron, T. Pierrot, N. Perrin, K. Beguir, and O. Sigaud, "QD-RL: Efficient mixing of quality and diversity in reinforcement learning," *arXiv preprint arXiv:2006.08505*, 2020.

[45] D. Gravina, A. Khalifa, A. Liapis, J. Togelius, and G. N. Yannakakis, "Procedural content generation through quality diversity," in *2019 IEEE Conference on Games (CoG)*. IEEE, 2019, pp. 1–8.

[46] A. G. Barto, "Intrinsic motivation and reinforcement learning," in *Intrinsically motivated learning in natural and artificial systems*. Springer, 2013, pp. 17–47.

[47] J. Schmidhuber, "Developmental robotics, optimal artificial curiosity, creativity, music, and the fine arts," *Connection Science*, vol. 18, no. 2, pp. 173–187, 2006.

[48] G. N. Yannakakis, "AI in computer games: generating interesting interactive opponents by the use of evolutionary computation," Ph.D. dissertation, University of Edinburgh, 2005.

[49] A. Liapis, G. N. Yannakakis, M. J. Nelson, M. Preuss, and R. Bidarra, "Orchestrating game generation," *IEEE Transactions on Games*, vol. 11, no. 1, pp. 48–68, 2018.