Finding Game Levels with the Right Difficulty in a Few Trials through Intelligent Trial-and-Error

Miguel González-Duque
IT University of Copenhagen
Copenhagen, Denmark
migd@itu.dk

Rasmus Berg Palm
IT University of Copenhagen
Copenhagen, Denmark
rasmb@itu.dk

David Ha
Google Brain
Tokyo, Japan
hadavid@google.com

Sebastian Risi
IT University of Copenhagen
Copenhagen, Denmark
sebr@itu.dk

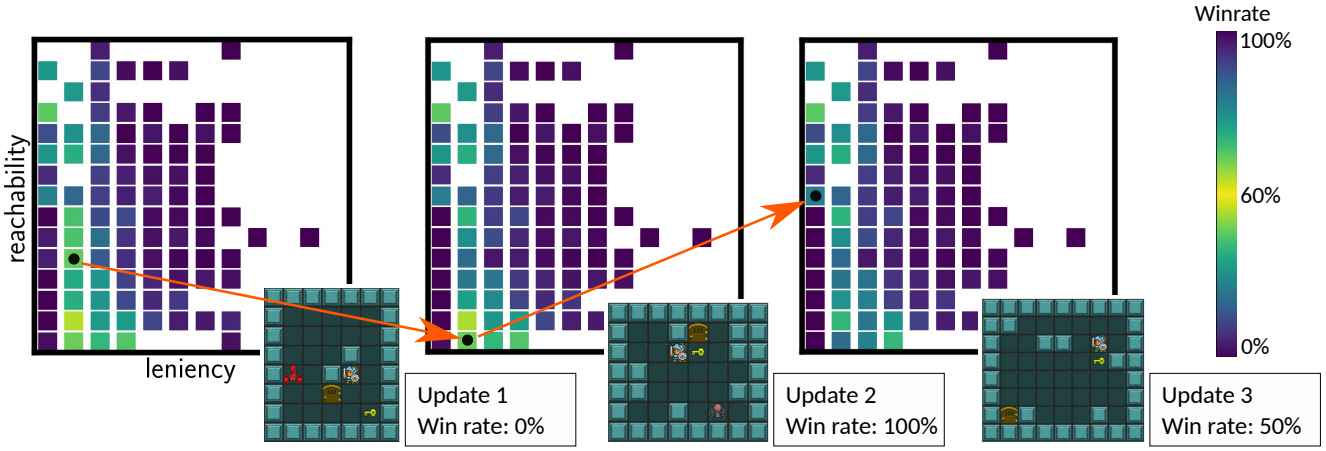


Fig. 1: Finding a level with the right difficulty via Intelligent Trial-and-Error (IT&E). IT&E [1] for games first creates a set of levels, arranged in a map that varies across level characteristics (amount of enemies and distance to goal). IT&E updates its beliefs about the difficulty of each level continuously using Gaussian Processes. In only three updates, the IT&E algorithm finds a level with ideal difficulty (win rate between 50%–70%) for a *One Step Look-Ahead* agent. The brighter the color in the maps, the closer the level is to the target difficulty, with darker colors representing levels that are either too easy or too hard.

***Abstract*—Methods for dynamic difficulty adjustment allow games to be tailored to particular players to maximize their engagement. However, current methods often only modify a limited set of game features such as the difficulty of the opponents, or the availability of resources. Other approaches, such as experience-driven Procedural Content Generation (PCG), can generate complete levels with desired properties such as levels that are neither too hard nor too easy, but require many iterations. This paper presents a method that can generate and search for *complete* levels with a specific target difficulty in only a few trials. This advance is enabled by through an *Intelligent Trial-and-Error algorithm*, originally developed to allow robots to adapt quickly. Our algorithm first creates a large variety of different levels that vary across predefined dimensions such as leniency or map coverage. The performance of an AI playing agent on these maps gives a proxy for how difficult the level would be for *another* AI agent (e.g. one that employs Monte Carlo Tree Search instead of Greedy Tree Search); using this information, a Bayesian Optimization procedure is deployed, updating the difficulty of the prior map to reflect the ability of the agent. The approach can reliably find levels with a specific target difficulty for a variety of planning agents in only a few trials, while maintaining an understanding of their skill landscape.**

***Index Terms*—Dynamic Difficulty Adjustment, Intelligent Trial-and-Error, Planning Agents, PCG, MAP-Elites**

I. Introduction

Dynamic Difficulty Adjustment (DDA) consists of adapting a game online, modifying its difficulty according to the behavior of the player [2]. With DDA, games try to maximize player engagement by continuously presenting challenges that are neither too easy, nor too difficult. If implemented properly, DDA promises to keep players engaged and entertained according to the psychological theory of *flow* [3].

Methods for DDA range from probability graphs [4], Monte Carlo Tree Search and other statistical forward planning algorithms [5], [6] to data-driven approaches [7]–[9]. However, these approaches come with two main drawbacks. First, they often modulate only a single aspect of a level or game (e.g. availability of resources [10], opponent AI [11]). Second, they do not capture and retain an understanding of what exactly is either causing the player trouble or boring them.

We develop a variation of the Bayesian-based Intelligent Trial-and-Error Algorithm (IT&E) to address these two challenges. First developed for fast adaptation in robots [1], this approach has the potential to both serve levels with the ideal level of difficulty and to maintain a player model that is constantly being updated using Bayesian Methods.

Another potential advantage of this approach (compared to e.g. experience-driven procedural content generation [12]) is its ability to find an ideal level faster.

In this paper we study the application of IT&E for DDA using *different* AI agents as a proxy for *different* human players. AI agents allow for a more clean-cut, controllable environment for testing our approach before applying it to real players. One of our hypotheses is that in a Bayesian optimization process, information on which levels one agent finds difficult can be used as an estimate of how difficult these levels will be for *another* agent. While this paper establishes the main method and shows that it works well for artificial agents, how this approach will scale to human players is an important future research direction.

II. Background

A. Dynamic Difficulty Adjustment

The idea behind Dynamic Difficulty Adjustment (DDA) is to modulate the difficulty of a game to keep the player in *flow* [3], a psychological state in which a task matches the ability of the user. One of the first approaches to DDA modulated Half-Life's internal economy using probabilistic methods based mostly on Inventory Theory [13]. Here, the current state of the game is mapped to an adjustment action that affects certain game aspects, such as the availability of health in a level [10]. Other approaches use probabilistic models to optimize player engagement and prevent churn in mobile games [4].

For Multi-player Online Battle Arena (MOBA) games, Silva et al. [11], [14] designed different AIs for easy, medium and hard difficulty, switching between them during deployment based on the performance difference between player and agent.

A data-driven approach for a turn-based role-playing game has been proposed by Zook & Riedl [8], which is based on a tensor factorization method to predict player performance.

Hao et al. [15] employed Monte Carlo Tree Search (MCTS) agents to modulate the difficulty of Pac-Man; by modifying the simulation time of the algorithm, the MCTS agents can be artificially handicapped to perform at different levels of difficulty. Demediuk et al. [5] expand on the idea of using a planning algorithm by introducing variations of MCTS that change the action selection policy or the heuristic for evaluating playouts.

However, these aforementioned approaches rely on adapting a limited set of game features to modulate difficulty. While approaches to overcome this limitation exist, allowing continuously adapting levels for a 2D platformer [7], they rely on an expensive mass data collection for estimating both level difficulty and player ability.

Our method evolves sets of entirely new and complete levels that exhibit several characteristics that can make the level easier or more difficult for different types of players. In contrast to previous approaches, here we maintain a set of evolved levels, updating our estimates of their difficulty. This way we are able to better understand which aspects of the level are challenging for the player and which are too easy (across the different behaviors that are measured).

B. Procedural Content Generation

The field of Procedural Content Generation (PCG) [16] is focused on creating game content algorithmically with little or no human intervention (e.g. game rules, characters, textures). This approach can benefit players by providing them unique experiences every time they play. Many PCG approaches rely on a fixed set of parameters and randomness to generate content within a heavily constrained space of possibilities, but a recent focus is to apply machine learning approaches to enable a more open-ended generation of content [17], [18].

In particular, PCG can be cast as a search problem, which tries to find content – such as levels – with particular properties [19]. For example, levels can be evolved to have a certain difficulty, to be balanced, and solvable. While this approach has the benefit of being very general and can create content as diverse as game maps [20], particle effects for weapons [21] or even game rules [22], searching for these artifacts can take many iterations. To limit the space of content the algorithm has to search through, these approaches can also be combined with self-supervised learning. *Latent Variable Evolution* (LVE) [23] is one such approach that has been applied successfully to creating artificial fingerprints [23] and Super Mario Bros. levels [24]. In LVE, a generator is trained on existing content and evolution only searches the space of latent variables given as input to the pre-trained generator. However, even in this case, search typically still takes many generations [24].

In contrast to existing search-based PCG work, the proposed approach can find levels with the right difficulty in only a few trials, by incorporating useful priors computed beforehand and using a sample efficient search. The algorithm is tested on generating new levels described in the Video Game Description Language (VGDL), which is explained next.

C. The GVGAI Framework

The General Video Game AI (GVGAI) framework provides a set of tools for testing artificial agents [25]. It was created to evaluate how well agents generalize over multiple games, which is the focus of the multiple competitions that are being held since its inception. GVGAI allows the user to easily generate and run games specified in the Video Game Description Langauge (VGDL) [26].

The IT&E for games approach introduced in this paper creates levels for a *Zelda*-like video game. The objective in this game consists of picking up a key and navigating towards the final goal. The level is arranged as a dungeon with enemies of three different kinds that move randomly at different speeds. The player loses the game if an enemy touches their avatar. The player is also equipped with a sword to kill opponents, which gives additional points.

The GVGAI framework comes with several game-playing agents that implement different planning algorithms. In this paper we test the algorithm to quickly find levels with the right difficulty for the following eight agents:

- **DoNothing**, an agent that stands still.
- **Random**, performs a random action at each timestep.

- **One Step Look-ahead** (OSLA), chooses the best next action according to a simple score heuristic that assesses the next states and picks the best performing one. This heuristic encourages the agent to kill opponents if they are one step away in the tree and to move towards whatever maximizes score if they are also at a one-step distance.
- **Greedy Tree Search** (GTS), searches the game tree for the next best state using the score as a heuristic. The search is limited by a time budget of 40ms per step imposed by the framework itself.
- **Random Search** (RS), creates a set of random playtraces from the current state up to a certain depth in the game tree (as much as the framework time budget allows), and follows the first step of the best playtrace according to a state heuristic that returns either the score or $\pm 10^6$ if the agent wins or loses respectively.
- **Rolling Horizon Evolution** (RHEA) [27] uses an evolutionary algorithm to compute an optimal playtrace up to a certain depth (with time budget of 40ms). The fitness function follows the same heuristic as in the RS setup.
- A vanilla implementation of the **Monte Carlo Tree Search** (MCTS) algorithm [28]. MCTS constructs an estimate of the average value of nodes in the game tree by running rollouts up to the end of the game. This value is guided by the same score heuristic that governs RS and RHEA, and the action selection takes into account an extra additive term that encourages exploration.
- **Open Loop Expectimax Tree Search** (OLETS), the agent that won the first edition of the single-player GVGAI competition [28]. At each step, OLETS runs a simulation in which it assesses the value of next actions according to the *Open Loop Expectimax* heuristic that includes not only the average value of the node but also the maximum value among its children. [29].

```
procedure MAP−Elites(n_iters, n_init):
    P = ∅
    X = ∅
    for iter = 1 → n_iters:
        if iter < n_init:
            x' = random_solution()
        else:
            x = random_selection(X)
            x' = random_variation(x)
        b' = behavior_descriptor(x')
        p' = performance(x')
        if P(b') = ∅ or P(b') < p':
            // update the elite in the cell
            P(b') = p'
            X(b') = x'
    return (P, X) // behavior−performance map
```

Algorithm 1: MAP-Elites' pseudocode.

D. Illumination Algorithms

While the goal of a typical evolutionary algorithm is to produce one solution that maximizes a given performance metric, the goal of illumination algorithms is to find high-performing solutions in different sections of the search space [30], [31]. By extending the search, illumination algorithms can find a plethora of individuals with high fitness that have different characteristics.

In this paper, we use *MAP-Elites* [30] (Algorithm 1). In MAP-Elites, the objective consists of obtaining an archive of elites, each expressing a different *behavior* in a low dimensional space. This behavior space is divided into cells, and each cell is identified by its centroid. Each cell maintains the best performing individual whose behavior is closest to its centroid. Normally, individuals are chosen uniformly at random from the current elites to be mutated. In the work here, MAP-Elites produces a map of levels that vary across certain dimensions such as the space covered or the distance to the goal.

Variants of the MAP-Elites algorithm have been applied to many game tasks before, such as evolving agents for game balancing in Hearthstone [32], mixed-initiative co-creation tools for dungeon level design [33], or level creation for Bullet Hell games [34]. To the best of our knowledge, our approach is the first one to leverage the combination of MAP-Elites and Bayesian Optimization (known as the Intelligent Trial-and-Error algorithm) for fast game difficulty adjustment.

E. Intelligent Trial-and-Error

Finding the optimal level of difficulty for a player can be thought of as an optimization process. Bayesian Optimization (BO) consists of learning a regression model for the function to be optimized and using it to drive the optimization [35]. The core idea of BO is to maintain a prior over all the possible forms of the objective function, and update it after querying using Bayes' rule. Bayesian regression is well known for being data-efficient and for working even for black-box functions.

The Intelligent Trial-and-Error algorithm (IT&E) [1] is a form of BO that relies on Gaussian Process Regression [36] and the MAP-Elites algorithm [30]. In Gaussian Process Regression, the objective is to learn a function $f(x)$ using an assumed prior function $\mu_0(x)$ and a kernel function $k(x, x')$ that describe the mean and covariance of the process respectively (we write $f(x) \sim \text{GP}(\mu_0(x), k(x, x'))$. A common choice for a kernel function k (which we will use for our experiments), is the Matérn$_{5/2}$ kernel, given by

$$k(x, x'; \sigma) = \sigma^2 \left(1 + \sqrt{5}r + \frac{5}{3}r^2\right) \exp\left(-\sqrt{5}r\right) \quad (1)$$

where r is the distance between x and x', and σ is a scalar hyperparameter [37]. By *conditioning* on newly arrived data $\boldsymbol{x} = [x_i]_{i=1}^t$ and $\boldsymbol{f} = [f_i]_{i=1}^t$, our estimation of $f(x)$ can be iteratively updated. If we want to predict $\tilde{f}$ for a particular $\tilde{x}$, we condition on $\boldsymbol{x}, \boldsymbol{f}$ and $\tilde{x}$ to get a normal distribution for $\tilde{f}$:

$$\begin{aligned} \tilde{f} \mid \tilde{x}, \boldsymbol{x}, \boldsymbol{f} \sim \mathcal{N}(&\mu_0(x) + \boldsymbol{k}(\tilde{x})^T K^{-1} \boldsymbol{f}, \\ &k(\tilde{x}, \tilde{x}) - \boldsymbol{k}(\tilde{x})^T K^{-1} \boldsymbol{k}(\tilde{x})), \end{aligned} \quad (2)$$

where $\boldsymbol{k}(\tilde{x}) = [k(\tilde{x}, x_i)]_{i=1}^t$, $K = [k(x_i, x_j)]_{i,j=1}^t + \sigma_{\text{noise}}^2 I$, and σ_{noise} is a hyperparameter.

Agent	Easy $1 \geq w \geq 0.8$	Medium $0.8 > w \geq 0.6$	Medium $0.6 > w \geq 0.4$	Medium $0.4 > w \geq 0.2$	Hard $0.2 > w \geq 0$
OLETS	326	2	1	0	0
MCTS	319	30	5	2	0
RHEA	246	82	13	1	0
RS	268	53	6	1	0
GTS	111	79	69	39	34
OSLA	60	17	22	14	220
Random	48	9	33	41	191
doNothing	0	0	0	0	341

TABLE I: Amount of levels per difficulty: Levels with win rate between 0.8 and 1 are easy for the agent, while levels with win rate between 0 and 0.2 can be considered hard. The *advanced* agents (OLETS, MCTS, RHEA and RS) find most levels easy; the map computed using Greedy Tree Search contains the most variety of difficulties, and the simpler controllers find most levels too difficult.

The IT&E algorithm starts by computing a prior map using MAP-Elites, and then uses Gaussian Processes to update these beliefs about the objective function. Let $\mathcal{P}(x)$ be the behavior-performance map (Algorithm 1). The algorithm first assigns $\mu_0(x) = \mathcal{P}(x)$, and then selects and deploys an elite $x_{t+1} \in \mathcal{X}$ by maximizing $\mu_t(x_{t+1}) + \beta\sigma_t(x_{t+1})$ where $\sigma_t(x_{t+1})$ is the posterior standard deviation. Once the real performance p_{t+1} (i.e. how difficult the map is for a new agent) is obtained, a new approximation of the performance map is computed with Equation 2. In other words, the algorithm starts by sampling the best performing x from the prior map constructed using MAP-Elites, and updates this map once the actual performance of x is received.

III. Approach: Fast Game Difficulty Adjustment through IT&E

In this paper, we use a planning agent P', as a proxy for a human player. We break down the task of quickly finding optimally difficult levels for P' into the following steps (Fig. 2): (**1**) Evolve a diverse set of levels with near-optimal win-rates for several AI agents (excluding P') using the MAP-Elites algorithm (Fig. 2a). Note: this step, although expensive, only needs to be done once and can be done offline. (**2**) Use the IT&E algorithm to quickly find an optimal level for P' (Fig. 2c), using the performance of *other* AI agents as a prior estimate of the performance of P' on each level (Fig. 2b).

In our implementation of IT&E, this process stops after finding an individual whose performance is above a certain predefined bound. The original IT&E implementation only stops after finding an individual whose performance is above $\alpha \max(\mu_t(x))$ for a hyperparameter $\alpha \in [0, 1]$, that is, when our estimates of the real performances indicate that there does not exist a better performing individual to be tested (up to a certain percentage governed by α).

A. Generating diverse and optimally difficult levels

Before we can run IT&E to quickly find a level with the right difficulty for a new agent (Section IV-A), we first have to create initial level maps for all the different AI-playing agents (Fig. 2a) through MAP-Elites (Algorithm 1), which is detailed in this section. The algorithm requires two functions:

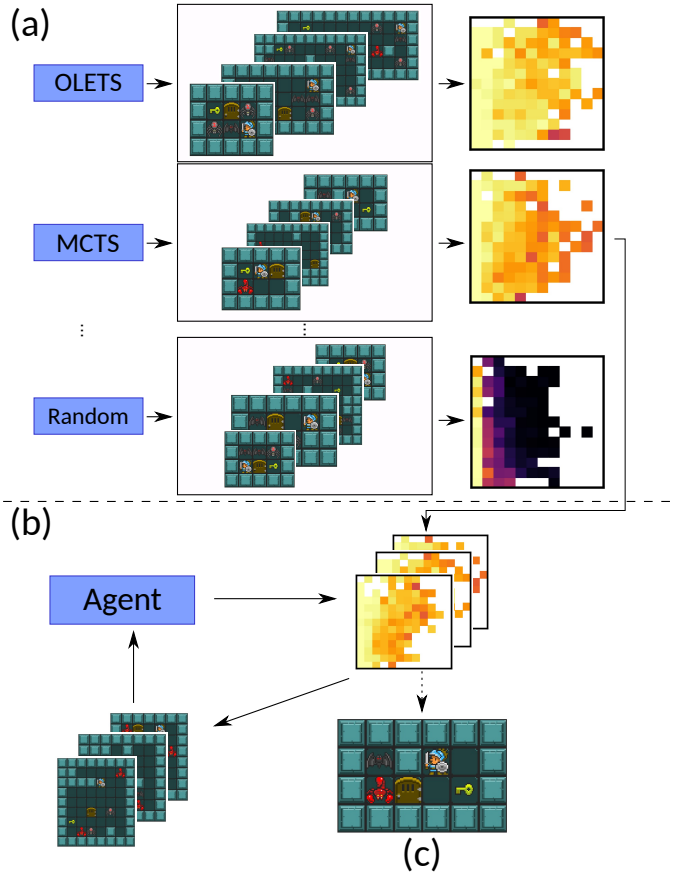
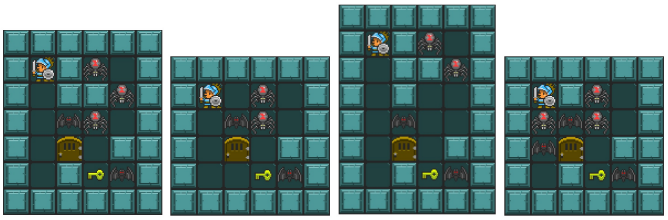


Fig. 2: An overview of our IT&E level generation approach. (a) First, a diverse set of levels are evolved such that their difficulty, evaluated as the win rate of several AI agents, is optimal. This set of diverse levels is organized in a map such that levels that have similar *behavioral* features (e.g. same amount of enemies, or level coverage) are close. Each cell in the map corresponds to a level, and the color represents how easy (bright) or difficult (dark) a level is for said agent. (b) These level difficulty maps are then used as priors in a Bayesian optimization procedure, which iterates over levels to quickly select an optimal level for *another* AI agent (c).

`random_solution()` which returns a random level and `random_variation(level)` which randomly mutates a level. The function `random_solution()` is implemented as follows:

1) Sample the width w and height h at random between 3 and 9.
2) Sample the amount of enemies e at random between $\lfloor \min(w, h)/2 \rfloor$ and $\min(w, h)$.
3) If $\min(w, h) > 3$, sample a random integer i, similarly to e, for the amount of inner walls in the level.[1]
4) If there is not enough room for placing the player, key, goal, enemies and walls, i.e. $i + e + 3 > (w - 2)(h - 2)$,

[1]Notice that if $\min(w, h) = 3$, then any inner wall would almost surely block the path between the avatar, the key, and the goal, making the game unwinnable.



(a) (b) (c) (d)

Fig. 3: Level creation and mutation: Shown are a generated example level (a) and three different level mutations (b–d). In (b) the third row and two walls were removed; in (c) a new row and a wall were added, and one enemy was removed; in (d) the third row was removed and two enemies were added.

adjust $h = h + 1$ or $w = w + 1$ (selecting at random) until there is enough room.

5) Create a level with width w and height h with walls at the borders.
6) Place the avatar, key, and goal at random in the level.
7) Repeating e times: randomly select a type of enemy and place it in a random available position.
8) Compute the A* path between the avatar and the key, and between the key and the goal. Mark the positions in this path as occupied.
9) If i is the number of inner walls and a is the number of available positions after removing the positions of the paths, then place $\min(a, i)$ inner walls at random.

The function `random_solution()` always returns a solvable level, because the A* path between avatar, key and goal are preserved. To mutate these levels, the function `random_variations(level)` takes a level and performs the following steps:

1) Expands or contracts the width/height by adding or removing one column/row at random (only considering the columns/rows that don't have either the avatar, key nor goal) if possible.
2) Adds or removes random enemies by sampling a random integer between -2 and 2, where negative numbers imply removing instead of adding.
3) Adds or removes inner walls similar to the previous step, verifying that the connectivity between Avatar, key, and goal is not broken.

With the two functions `random_solution()` and `random_variation(level)` we can create new levels and mutate them (Fig. 3).

The performance function should drive MAP-Elites towards creating a map of levels that are neither too easy nor too hard for the particular AI agent. Choosing the win rate itself as the value to be optimized would drive the process towards creating maps with levels that are too easy (i.e. levels in which the win rate is 100%). Therefore we define a performance metric p that has a maximum of 1 when the agent performs at win rates of 60% and goes down to 0 at both 0% and 100% win rate (Fig. 4). With this custom fitness function, we are encouraging

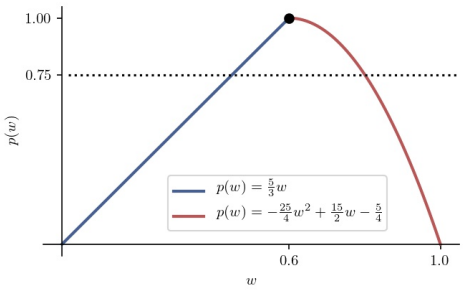


Fig. 4: Performance function: since using win rate alone would encourage the creation of only easy levels, we used a custom performance function that has its maximum at 60% win rate. The rationale for choosing 60% win rate is that easy levels (with win rates closer to 100%) would be boring for the player, and levels that are too difficult (with win rates closer to 0%) would be frustrating. This function is defined as $p(w) = (5/3)w$ up to 0.6 and as $p(w) = -(25/4)w^2 + (15/2)w - 5/4$ from 0.6 to 1. When computing the ideal level for an agent using a different prior (see Subsec. IV-A), we stopped after finding a level such that $p(w) \geq 0.75$.

the creation of levels with 60% win rate. Performance for each agent deployed in a level is measured as the average across 40 rollouts. For each generated level, we compute the following behavioral features that determined its location in the map:

- **Space coverage:** the percentage of the level that is filled with prompts (how *crowded* the level is).
- **Leniency:** the number of enemies in the level.
- **Reachability:** the sum of distances of the paths between the avatar and the key and the key and the goal.

To summarise, at the end of the map generation process we have different maps of elites for the different agents, together with average win rates for each level.

IV. Results

For each agent, MAP-Elites was run for 10 generations, with an initialization of 100 levels and with 50 iterations per generation after that (Fig. 5). Since the behavioral features are computed in $\mathbb{R}^3$, maps show a 2D projection obtained by averaging over the remaining behavioral feature. An example of an evolved level with a win rate of approximately 60% is also shown.

These maps reflect the aptitudes and deficiencies of each of the different agents employed in the experiments. The OLETS agent finds most levels too easy (for the behavioral features that were measured). This observation is supported by the results in Table I, which show a count of levels, segmented by difficulty. This is also the case for most of the *advanced* planning agents we investigated (MCTS, RHEA and RS).

On the other hand, the Random agent only finds agents without opponents easy, since it will eventually stumble upon the key and the goal. From the basic agents, Greedy Tree Search (GTS) is better able to deal with sparsity and longer distances

Prior\Agent	OLETS	MCTS	RHEA	RS	GTS	OSLA	Random
OLETS	1.3 (10/10)	**1.2 (10/10)**	1.4 (10/10)	1.6 (10/10)	7.1 (9/10)	(0/10)	(0/10)
MCTS	11.7 (7/10)	1.2 (10/10)	2.1 (10/10)	2.9 (10/10)	6.7 (10/10)	11.7 (10/10)	(0/10)
RHEA	(0/10)	2.5 (10/10)	1.1 (10/10)	**1.0 (10/10)**	3.2 (10/10)	15.6 (7/10)	(0/10)
RS	**5.4 (8/10)**	2.0 (10/10)	**1.1 (10/10)**	1.5 (10/10)	3.7 (10/10)	12.0 (10/10)	(0/10)
GTS	(0/10)	11.6 (9/10)	7.1 (10/10)	7.5 (8/10)	1.1 (10/10)	11.7 (3/10)	2.6 (10/10)
OSLA	(0/10)	10.5 (10/10)	3.8 (10/10)	5.8 (10/10)	5.8 (10/10)	1.2 (10/10)	20.0 (2/10)
Random	(0/10)	(0/10)	(0/10)	(0/10)	13.7 (3/10)	**3.5 (10/10)**	1.3 (10/10)
doNothing	(0/10)	8.8 (6/10)	3.0 (1/10)	12.0 (3/10)	**2.4 (10/10)**	9.5 (2/10)	11.4 (5/10)
Baseline (noise)	(0/10)	15.7 (7/10)	3.9 (10/10)	4.5 (10/10)	**1.0 (10/10)**	(0/10)	**2.0 (10/10)**

TABLE II: Mean iterations to find a level with ideal difficulty. This table presents the average number of iterations to find a compensatory level for all pairs of priors and agents. It also presents the amount of successful adaptions (e.g. 7/10 means that the algorithm was able to find a level with win rate in approximately [0.5, 0.7] in 7 out of 10 runs). This average was computed only on successful runs. These results show that we can find an ideal level of difficulty for an agent using the prior of *another* after a few updates. There is a relationship between the skill of the agent and the quality of the map (i.e. the skill of the agent that was used to create it). Deploying IT&E using the map of a skilled agent on a more basic one translates to more updates, and vice versa. This makes intuitive sense: the levels in the archive map of a skilled agent might be *too difficult* for a more basic agent, and it takes more iterations to find a level that is *easy enough*.

to the goals. This result can be explained by the fact that GTS can search deeper into the game tree. The exact opposite is true for the One Step Look-Ahead agent (OSLA), which can only deal with levels in which the objectives are close-by (low *reachability*). The presence or absence of enemies (measured by *leniency*) has no impact on the performance of OSLA.

These insights are also reflected by the high-performing levels (i.e. win rate close to 60%; see Fig. 4) shown in Fig. 5. OLETS successfully navigates levels even when surrounded by enemies, and the lower performing agents can only solve levels with fewer enemies. GTS can navigate to further goals (e.g. the key on the lower right), while the Random agent struggles even when it is close to the objectives.

A. Dynamic Difficulty Adjustment through IT&E

After computing all the elite archives, we test whether the IT&E can quickly provide a level of appropriate difficulty if the agent changed starting from a particular performance map. For example, would the priors of a map created through an MCTS agent allow faster adaptation for an OLETS agent than priors from a DoNothing agent?

We deploy a variation of the original IT&E algorithm [1] that stops after finding a compensatory level with win rate approximately between 0.5 and 0.7. Expressing this win rate in terms of performance function (Fig. 4), means finding a level with performance $p \geq 0.75$.

The experiment of searching for the optimal level was repeated ten times for each pair of agents. Since the goal is to test for fast adaption, each experiment was stopped if the algorithm did not find a compensatory level in the first 20 updates. In that case, the search is counted as unsuccessful. For our experiments, we choose a Matérn$_{5/2}$ kernel with lengthscale $\sigma = 1$ (Eq. 1), noise variance given by $\sigma^2_{\text{noise}} = 0.1$, and $\beta = 0.03$ (Sec. II-E).

In addition to the agents discussed in the previous section, a baseline behavior map is constructed by assigning random performances to the cells of the DoNothing prior (i.e. a random selection of levels). Running an experiment with this baseline is akin to choosing a random ordering off all the levels in the archive of the DoNothing agent, starting with the first one, and running continuous updates thereafter. With this baseline, we aim to measure whether the priors are useful for quick adaptation, compared to just sampling levels at random.

Table II shows the mean number of iterations it took IT&E to find a level with the right amount of difficulty for all successful runs, together with the number of successful runs. Given the prior of *another* agent, the algorithm can find a level in only a few iterations (at most 16) for all agents. The skill (or lack thereof) of certain agents make it difficult to find a compensatory level: the OLETS agent, in particular, performs at such a high level that the priors of more basic agents have no level difficult enough; the opposite is true for the random agent, which performs so poorly that the advanced agent priors have no easy enough levels.

Fig. 1 shows an example of the IT&E adaptation steps. Starting with the Random agent archive as a prior, the algorithm finds a level that is difficult enough for the One Step Look-Ahead (OSLA) agent after 3 updates. The heatmap shows performance $p(w)$ instead of win rate, and brighter colors represent levels that are close to the ideal difficulty of 60%. The first level selected is the best performing one for the Random agent (Fig. 5), and the OSLA agent loses all 40 rollouts. Since this level is too difficult, the search is moved towards a level in which the key and goal are one step away from each other, achieving 100% win rate. Because the stopping criterion has not been achieved, the algorithm continues and in the next iteration finds a level that is quite similar to the ideal level in the OSLA prior (Fig. 5). This

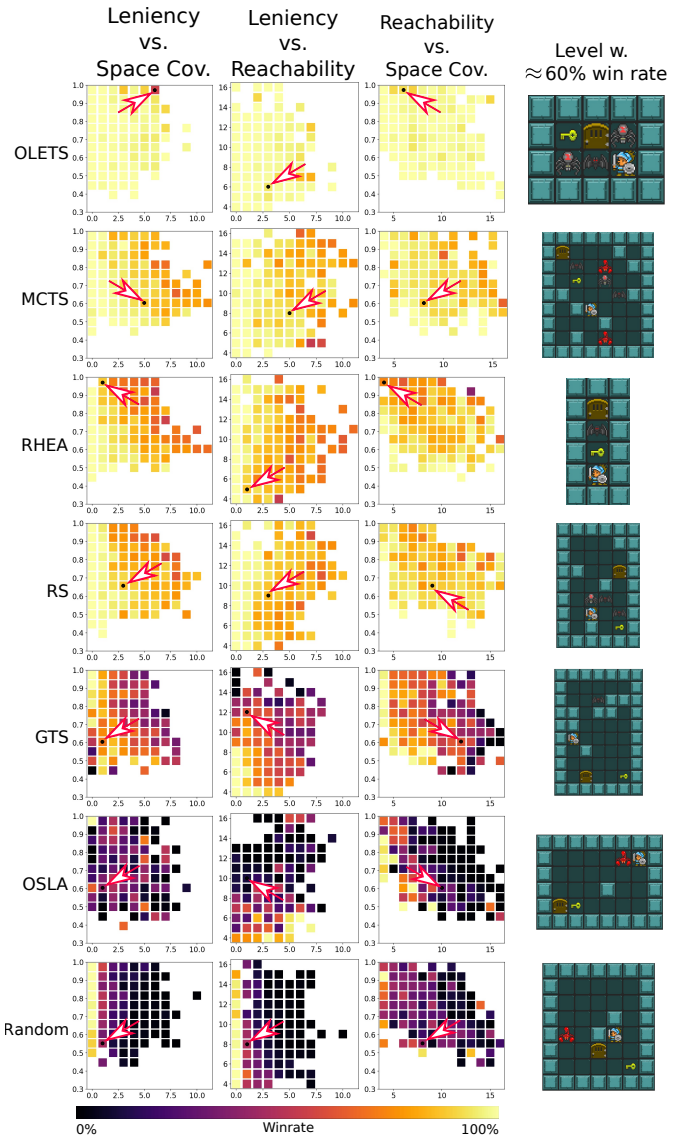


Fig. 5: Archive of elites for different agents: Shown are the final generation of the behavior map obtained for different planning agents, colored by win rate. The brighter the color, the higher the win rate for the agent. The position of a level with a particular win rate (60%) is highlighted in the map. These maps illustrate what each of the agents finds easy or difficult. For example, the OSLA agent struggles at levels with higher reachability (i.e. longer distances to the key and goal), because it only looks at the next nodes in the game tree.

level has approximately the ideal difficulty for OSLA since it performs at 50% win rate. Since color represents how close a level is to optimal difficulty, it is worth asking why the maps seem to select levels in cells that are not the brightest. This can be explained by two reasons: the acquisition function in the Gaussian Process encourages exploration (Sec. II-E), and, since these maps are a projection into two dimensions obtained by averaging over the remaining behavioral feature, the behavior selected may be the highest performing one. This last fact also explains why the changes in the map seem to be small when they actually are not.

A surprising result in these experiments is that the baseline prior (which consists of a random assignment of performance to the levels in the doNothing prior) serves as a good prior for the IT&E algorithm in most agents. We argue that this is happening because of two reasons. First, in this random ordering of levels in the baseline prior, the best performing levels coincide by chance with the ideal levels for Greedy Tree Search and the Random agent; secondly, what makes the IT&E algorithm work lies more in the Bayesian update component than in the quality of the map. This last fact has also been discussed in other applications of the IT&E algorithm [38]. However, using a good prior still leads to faster difficulty adjustment in most cases.

V. Discussion and Conclusion

In this article we tested whether the Intelligent Trial-and-Error algorithm (IT&E) [1] could be used for Dynamic Difficulty Adjustment (DDA); for this, we evolved several archives of difficult levels for different planning agents and lay them in a *behavior map* using MAP-Elites [30], and we used these maps as priors in a Gaussian Process. We were able to find levels with ideal difficulty for an agent using the archive of *another* agent after a few updates.

These behavior maps reflect the aptitudes and deficiencies of the agents that drove their evolution. The Random agent's archive shows better performance when there are no enemies in the level, while higher-performing planning agents such as Monte Carlo Tree Search (MCTS) are able to deal with opponents (Fig. 5).

When deploying the IT&E algorithm, the disparity between the quality of the prior map (that is, the skill of the *other* agent) and the abilities of the agent has an impact on how quickly an ideal level is found. This implies that, for our approach to work well for human players, we would need agents that perform at a level comparable to them, which is feasible for many simpler games. In games where this is not feasible, human playtraces could be used to either build useful prior maps directly, or train humanlike agents, through e.g. imitation learning.

These results, and our methodology, leave room for discussion. The use of hand-crafted behavioral features in MAP-Elites comes at the risk of selecting the *wrong ones* (that is, where maps that should exhibit similar difficulties lie further apart); in future work, the use of unsupervised feature learning (e.g. using Variational Autoencoders [17], [39]) could be explored. Moreover, as in [40], the most useful agent prior could be selected in an online fashion, i.e. during play. Furthermore, the proposed method can only select from the levels found by MAP-Elites. In future research, we plan to extend this to generating levels online, using the Gaussian Process to estimate the difficulty of the level.

Since our method can robustly model what the agent finds difficult about the levels that are being served, we believe our method has the potential to perform Dynamic Difficulty

Adjustment in domains in which understanding the player's abilities is useful (e.g. education & rehabilitation games).

ACKNOWLEDGEMENTS

We thank Julian Togelius for helpful discussions. SR thanks for support by Google (Google Faculty Research Award 2019). Funding for this research was provided by the Danish Ministry of Education and Science, Digital Pilot Hub and Skylab Digital.

REFERENCES

[1] A. Cully, J. Clune, D. Tarapore, and J.-B. Mouret, "Robots that can adapt like animals," *Nature*, vol. 521, no. 7553, pp. 503–507, 2015.

[2] M. Zohaib, "Dynamic difficulty adjustment (DDA) in computer games: A review," *Advances in Human-Computer Interaction*, vol. 2018, 2018.

[3] M. Csikszentmihalyi, *Flow: The Psychology of Optimal Experience*. New York, NY: Harper Perennial, March 1991.

[4] S. Xue, M. Wu, J. Kolen, N. Aghdaie, and K. A. Zaman, "Dynamic difficulty adjustment for maximized engagement in digital games," *26th International World Wide Web Conference 2017, WWW 2017 Companion*, pp. 465–471, 2017.

[5] S. Demediuk, M. Tamassia, W. L. Raffe, F. Zambetta, X. Li, and F. Mueller, "Monte Carlo tree search based algorithms for dynamic difficulty adjustment," *2017 IEEE Conference on Computational Intelligence and Games, CIG 2017*, pp. 53–59, 2017.

[6] S. Demediuk, M. Tamassia, X. Li, and W. L. Raffe, "Challenging AI: Evaluating the Effect of MCTS-Driven Dynamic Difficulty Adjustment on Player Enjoyment," *ACM International Conference Proceeding Series*, 2019.

[7] M. Jennings-Teats, G. Smith, and N. Wardrip-Fruin, "Polymorph: Dynamic Difficulty Adjustment through level generation," *Workshop on Procedural Content Generation in Games, PC Games 2010, Co-located with the 5th International Conference on the Foundations of Digital Games*, no. June 2010, pp. 2–6, 2010.

[8] A. E. Zook and M. O. Riedl, "A temporal data-driven player model for dynamic difficulty adjustment," *Proceedings of the 8th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2012*, pp. 93–98, 2012.

[9] A. Zook, S. Lee-Urban, M. R. Drinkwater, and M. O. Riedl, "Skill-based mission generation: A data-driven temporal player modeling approach," *3rd Workshop on Procedural Content Generation in Games, PCG 2012, Organized in Conjunction with the Foundations of Digital Games Conference, FDG 2012*, pp. 78–85, 2012.

[10] R. Hunicke, "The case for dynamic difficulty adjustment in games," *ACM International Conference Proceeding Series*, vol. 265, pp. 429–433, 2005.

[11] M. P. Silva, V. do Nascimento Silva, and L. Chaimowicz, "Dynamic difficulty adjustment on MOBA games," *Entertainment Computing*, vol. 18, pp. 103–123, 2017.

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

[13] R. Hunicke and V. Chapman, "AI for dynamic difficulty adjustment in games," *AAAI Workshop - Technical Report*, vol. WS-04-04, pp. 91–96, 2004.

[14] M. P. Silva, V. d. N. Silva, and L. Chaimowicz, "Dynamic difficulty adjustment through an adaptive ai," in *14th Brazilian Symposium on Comp. Games and Dig. Entertainment (SBGames)*, 2015, pp. 173–182.

[15] Y. Hao, Suoju He, Junping Wang, Xiao Liu, jiajian Yang, and Wan Huang, "Dynamic difficulty adjustment of game ai by mcts for the game pac-man," in *2010 Sixth International Conference on Natural Computation*, vol. 8, 2010, pp. 3918–3922.

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

[17] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgard, 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.

[18] S. Risi and J. Togelius, "Increasing generality in machine learning through procedural content generation," *arXiv preprint arXiv:1911.13071*, 2019.

[19] 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.

[20] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelbäck, G. N. Yannakakis, and C. Grappiolo, "Controllable procedural map generation via multiobjective evolution," *Genetic Programming and Evolvable Machines*, vol. 14, no. 2, pp. 245–277, 2013.

[21] E. J. Hastings, R. K. Guha, and K. O. Stanley, "Automatic content generation in the galactic arms race video game," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 4, pp. 245–263, 2009.

[22] J. Togelius and J. Schmidhuber, "An experiment in automatic game design," in *2008 IEEE Symposium On Computational Intelligence and Games*. IEEE, 2008, pp. 111–118.

[23] P. Bontrager, A. Roy, J. Togelius, N. Memon, and A. Ross, "Deepmasterprints: Generating masterprints for dictionary attacks via latent variable evolution," in *2018 IEEE 9th International Conference on Biometrics Theory, Applications and Systems (BTAS)*. IEEE, 2018, pp. 1–9.

[24] 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.

[25] D. Perez-Liebana, S. Samothrakis, J. Togelius, S. M. Lucas, and T. Schaul, "General video game AI: Competition, challenges, and opportunities," *30th AAAI Conference on Artificial Intelligence, AAAI 2016*, pp. 4335–4337, 2016.

[26] T. Schaul, "A video game description language for model-based or interactive learning," in *Proceedings of the IEEE Conference on Computational Intelligence in Games*. Niagara Falls: IEEE Press, 2013.

[27] D. Perez, S. Samothrakis, S. M. Lucas, and P. Rohlfshagen, "Rolling horizon evolution versus tree search for navigation in single-player real-time games," *Proceedings of the 2013 Genetic and Evolutionary Computation Conference*, pp. 351–358, 2013.

[28] 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, vol. 3, no. 2, https://gaigresearch.github.io/gvgaibook/.

[29] D. Perez-Liebana, S. Samothrakis, J. Togelius, T. Schaul, and S. M. Lucas, "Analyzing the robustness of general video game playing agents," *IEEE Conference on Computatonal Intelligence and Games, CIG*, vol. 0, 2016.

[30] J.-B. Mouret and J. Clune, "Illuminating search spaces by mapping elites," *arXiv preprint arXiv:1504.04909*, 2015.

[31] J. Lehman and K. O. Stanley, "Evolving a diversity of virtual creatures through novelty search and local comp," *Proc. of 13th Genetic & Evol. Comp. Conf*, no. Gecco, pp. 211–218, 2011.

[32] M. C. Fontaine, F. De Mesentier Silva, S. Lee, J. Togelius, L. B. Soros, and A. K. Hoover, "Mapping hearthstone deck spaces through map-elites with sliding boundaries," *GECCO 2019 - Proceedings of the 2019 Genetic and Evolutionary Computation Conference*, pp. 161–169, 2019.

[33] A. Alvarez, S. Dahlskog, J. Font, and J. Togelius, "Empowering quality diversity in dungeon design with interactive constrained MAP-Elites," *IEEE Conference on Computatonal Intelligence and Games, CIG*, vol. 2019-Augus, 2019.

[34] A. Khalifa, A. Nealen, S. Lee, and J. Togelius, "Talakat: Bullet hell generation through constrained map-elites," *GECCO 2018 - Proceedings of the 2018 Genetic and Evolutionary Computation Conference*, pp. 1047–1054, 2018.

[35] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, "Taking the human out of the loop: A review of bayesian optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2016.

[36] C. E. Rasmussen, *Gaussian Processes in Machine Learning*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 63–71.

[37] GPy, "GPy: A gaussian process framework in python," http://github.com/SheffieldML/GPy, since 2012.

[38] D. Tarapore, J. Clune, A. Cully, and J. B. Mouret, "How do different encodings influence the performance of the MAP-Elites algorithm?" *GECCO 2016 - Proceedings of GECCO 2016*, pp. 173–180, 2016.

[39] S. Thakkar, C. Cao, L. Wang, T. J. Choi, and J. Togelius, "Autoencoder and evolutionary algorithm for level generation in lode runner," *IEEE Conference on Computatonal Intelligence and Games, CIG*, vol. 2019-August, 2019.

[40] R. Kaushik, P. Desreumaux, and J.-B. Mouret, "Adaptive prior selection for repertoire-based online adaptation in robotics," *Frontiers in Robotics and AI*, vol. 6, p. 151, 2020.