Evolving Game Skill-Depth using General Video Game AI Agents

Jialin Liu
University of Essex
Colchester, UK
jialin.liu@essex.ac.uk

Julian Togelius
New York University
New York City, US
julian.togelius@nyu.edu

Diego Pérez-Liébana
University of Essex
Colchester, UK
dperez@essex.ac.uk

Simon M. Lucas
University of Essex
Colchester, UK
sml@essex.ac.uk

***Abstract*—Most games have, or can be generalised to have, a number of parameters that may be varied in order to provide instances of games that lead to very different player experiences. The space of possible parameter settings can be seen as a search space, and we can therefore use a Random Mutation Hill Climbing algorithm or other search methods to find the parameter settings that induce the best games. One of the hardest parts of this approach is defining a suitable fitness function. In this paper we explore the possibility of using one of a growing set of General Video Game AI agents to perform automatic play-testing. This enables a very general approach to game evaluation based on estimating the skill-depth of a game. Agent-based play-testing is computationally expensive, so we compare two simple but efficient optimisation algorithms: the Random Mutation Hill-Climber and the Multi-Armed Bandit Random Mutation Hill-Climber. For the test game we use a space-battle game in order to provide a suitable balance between simulation speed and potential skill-depth. Results show that both algorithms are able to rapidly evolve game versions with significant skill-depth, but that choosing a suitable resampling number is essential in order to combat the effects of noise.**

***Index Terms*—Automatic game design, game tuning, optimisation, RMHC, GVG-AI**

I. Introduction

Designing games is an interesting and challenging discipline traditionally demanding creativity and insight into the types of experience which will cause players to enjoy the game or at least play it and replay it. There have been various attempts to automate or part-automate the game generation process, as this is an interesting challenge for AI and computational creativity [1], [2], [3]. So far the quality of the generated games (with some exceptions) do not challenge the skill of human game designers. This is because the generation of complete games is a more challenging task than the more constrained task of generating game content such as levels or maps. Many video games require content to be produced for them, and recent years have seen a surge in AI-based procedural content generation [4].

There is another aspect of AI-assisted game design which we believe is hugely under-explored: automatic game tuning. This involves taking an existing game (either human-designed or auto-generated) and performing a comprehensive exploration of the parameter space to find the most interesting game instances.

Recent work has demonstrated the potential of this approach, automatically generating distinct and novel variants of the minimalist mobile game *Flappy Bird* [5]. That work involved using a very simple agent to play through each generated game instance. Noise was added to the selected actions, and a game variant was deemed to have an appropriate level of difficulty if a specified number of players achieved a desired score. For Flappy Bird it is straightforward to design an AI agent capable of near-optimal play. Adding noise to the selected actions of this player can be used to provide a less than perfect agent that better represents human reactions. An evolutionary algorithm was used to search for game variants that were as far apart from each other in parameter space as possible but were still playable.

However, for more complex games it is harder to provide a good AI agent, and writing a new game playing agent for each new game would make the process more time consuming. Furthermore, a single hand-crafted agent may be blind to novel aspects of evolved game-play elements that the designer of the AI agent had not considered. This could severely inhibit the utility of the approach. In this work we mitigate these concerns by tapping in to an ever-growing pool of agents designed for the General Video Game AI (GVG-AI) competition[1]. The idea is that using a rich set of general agents will provide the basis for a robust evaluation process with a higher likelihood of finding skill-depth wherever it may lie in the chosen search space of possible games. In this paper we use one of the sample GVG-AI agents, varying it by changing the rollout budget. This was done by making the game implement a standard GVG-AI game interface, so that any GVG-AI agent can be used with very little effort, allowing the full set of agents to be used in future experiments.

Liu et al. [6] introduced a two-player space-battle game, derived from the original *Spacewar*, and performed a study on different parameter settings to bring out some strengths and weaknesses of the various algorithms under test. A key finding is that the rankings of the algorithms depend very much on the details of the game. A mutation of one parameter may lead to a totally different ranking of algorithms. If the game using only a single parameter setting is tested, the conclusions could be less robust and misleading.

[1] http://www.gvgai.net/

978-1-5090-4601-0/17/$31.00

In this paper, we adapt the space-battle game introduced by Liu et al. [6] to the GVG-AI framework, then uses the Random Mutation Hill Climber (RMHC) and Multi-Armed Bandit RMHC (MABRMHC) to evolve game parameters to provide some game instances that lead to high winning rates for GVG-AI sample MCTS agents. This is used as an approximate measure of skill-depth, the idea being that the smarter MCTS agents should beat unintelligent agents, or that MCTS agents with a high rollout budget should beat those with a low rollout budget.

The paper is structured as follows: Section II provides a brief review of the related work on automatic game design, Section III describes the game engine, Section IV introduces the two optimisation algorithms used in this paper, Section V presents the experimental results, finally Section VI concludes and discusses the potential directions in the future.

II. Automatic game design and depth estimation

Attempts to automatically design complete games go back to Barney Pell, who generated rules for chess-like games [7]. It did not however become an active research topic until the late 2000's.

Togelius et al. [8] evolved racing tracks in a car racing game using a simple multi-objective evolutionary algorithm called *Cascading Elitism*. The fitness functions attempted to capture various aspects of player experience, using a neural network model of the player. This can be seen as an early form of experience-driven procedural content generation [9], where game content is generated through search in content space using evolutionary computation or some other form of stochastic optimisation. Similar methods have since been used to generate many types of game content, such as particle systems for weapons in a space shooter [10], platform game levels [11] or puzzles [12]. In most of these cases, the fitness functions measure some aspect of problem difficulty, with the assumption that good game content should not make the game too hard nor too easy.

While the research discussed above focuses on generating content for an existing game, there have been several attempts to use the search-based methods to generate new games by searching though spaces of game rules. Togelius and Schmidhuber [1] used a simple hill-climber to generate single-player Pac-Man-like games given a restricted rule search space. The fitness function was based on learnability of the game, operationalised as the capacity of another machine learning algorithm to learn to play the game.

This approach was taken further by Cook et al. [13], [3], who used search-based methods to design rulesets, maps and object layouts in tandem for producing simple arcade games via a system called ANGELINA. Further iterations of this system include the automatic selection of media sources, such as images and resources, giving this work a unique flavour. In a similar vein, Browne and Maire [2] developed a system for automatic generation of board games; they also used evolutionary algorithms, and a complex fitness function based on data gathered from dozens of humans playing different board games. Browne's work is perhaps the only to result in a game of sufficient quality to be sold as a stand-alone product; this is partly a result of working in a constrained space of simple board games.

A very different approach to game generation was taken by Nelson and Mateas [14], who use reasoning methods to create Wario Ware-style minigames out of verb-noun relations and common minigame design patterns. Conceptnet and Wordnet were used to find suitable roles for game objects.

Quite recently, some authors have used search-based methods to optimise the parameters of a single game, while keeping both game rules and other parts of the game content constant. In the introduction we discussed the work of Isaksen et al. on generating playable *Flappy Bird* variants [5]. Similarly, Powley et al. [15] optimise the parameters of an abstract touch-based mobile game, showing that parameter changes to a single ruleset can give rise to what feels and plays like different games.

One of the more important properties of a game can be said to be its *skill depth*, often just called *depth*. This property is universally considered desirable by game designers, yet it is hard to define properly; some of the definitions build on the idea of a skill chain, where deeper games simply have more things that can be learned [16]. Various attempts have been made to algorithmically estimate depth and use it as a fitness function; some of the research discussed above can be said to embody an implicit notion of depth in their fitness functions. Relative Algorithm Performance Profiles (RAPP) is a more explicit attempt at game depth estimation; the basic idea is that in a deeper game, a better player get relatively better result than a poorer player. Therefore, we can use game-playing agents of different strengths to play the same game, and the bigger the difference in outcome the greater the depth [17].

In this paper we use a form of RAPP to try to estimate the depth of variants of a simple two-player game. Using this measure as a fitness function, we optimise the parameters of this game to try to find deeper game variants, using two types of Random-Mutation Hill-Climber. The current work differs from the work discussed above both in the type of game used (two-player physics-based game), the search space (a multi-dimensional discrete space) and the optimisation method. In particular, compared to previous work by Isaksen et al, the current paper investigates a more complex game and uses a significantly more advanced agent, and also optimizes for skill-depth rather than difficulty. This work is, as far as we know, the first attempt to optimize skill-depth that has had good results.

III. Framework

We adapt the two-player space-battle game introduced by Liu et al. [6] to the GVG-AI framework, then use RMHC and MABRMHC to evolve game parameters to provide some game instances that lead to high winning rate for GVG-AI sample MCTS agents. The main difference in the modified space-battle game used in this work is the introduction of weapon system. Each ship has the choice to fire a missile after its cooldown period has finished. From now on, we use the term

"game" to refer to a game instance, i.e. a specific configuration of game parameters.

a) Spaceship: Each player/agent controllers a spaceship which has a maximal speed, v_s units distance per game tick, and slows down over time. At each game tick, the player can choose to *do nothing* or to make an action among $\{$*RotateClockwise, RotateAnticlockwise, Thrust, Shoot*$\}$. A missile is launched while the *Shoot* action is chosen and its cooldown period is finished, otherwise, no action will be taken (like *do nothing*). The spaceship is affected by a random recoil force when launching a missile.

b) Missile: A missile has a maximal speed, v_m units distance per game tick, and vanishes into nothing after 30 game tick. It never damages its mother ship.

Every spaceship has a radius of 20 pixels and every missile has a radius of 4 pixels in a layout of size 640*480.

c) Score: Every time a player hits its opponent, it obtains 100 points (reward). Every time a player launches a missile, it is penalized by c points (cost). Given a game state s, the player $i \in \{1, 2\}$ has a $score$ calculated by:

$$score(i) = 100 \times nb_k(i) - c \times nb_m(i), \quad (1)$$

where $nb_k(i)$ is the number of lives subtracted from the opponent and $nb_m(i)$ indicates the number of launched missiles by player $i \in \{1, 2\}$.

d) End condition: A game ends after 500 game ticks. A player wins the game if it has higher score than its opponent after 500 game ticks, and it's a loss of the other player. If both players have the same score, it's a draw.

e) Parameter space: The parameters to be optimised are detailed in Table I. There are in total 14,400 possible games in the 5-dimensional search space. Fig. 1 illustrates briefly how the game changes by varying only the cooldown time for firing missiles.

TABLE I
GAME PARAMETERS. ONLY THE FIRST 5 PARAMETERS ARE OPTIMISED IN THE PRIMARY EXPERIMENTS. THE LAST ONE (SHIP RADIUS) IS TAKEN INTO ACCOUNT IN SECTION V-C.

Parameter	Notation	Legal values	Dimension
Maximal ship speed	v_s	$4, 6, 8, 10$	4
Thrust speed	v_t	$1, 2, 3, 4, 5$	5
Maximal missile speed	v_m	$1, 2, 3, 4, 5, 6, 7, 8, 9, 10$	10
Cooldown time	d	$1, 2, 3, 4, 5, 6, 7, 8, 9$	9
Missile cost	c	$0, 1, 5, 10, 20, 50, 75, 100$	8
Ship radius	sr	$10, 20, 30, 40, 50$	5

The game is stochastic but fully observable. Each game starts with the agents in the symmetric positions. The two agents make simultaneous moves and in a fair situation. Thus, changing the player id does not change the situation of any player.

IV. OPTIMISERS

We compare a Random Mutation Hill-Climber to an Multi-Armed Bandit Random Mutation Hill-Climber in evolving instances for space-battle game described previously. This section is organised as follows. Section IV-A briefly recalls the Random Mutation Hill-Climber. Section IV-B presents the Multi-Armed Bandit Random Mutation Hill-Climber and its selection and mutation rules.

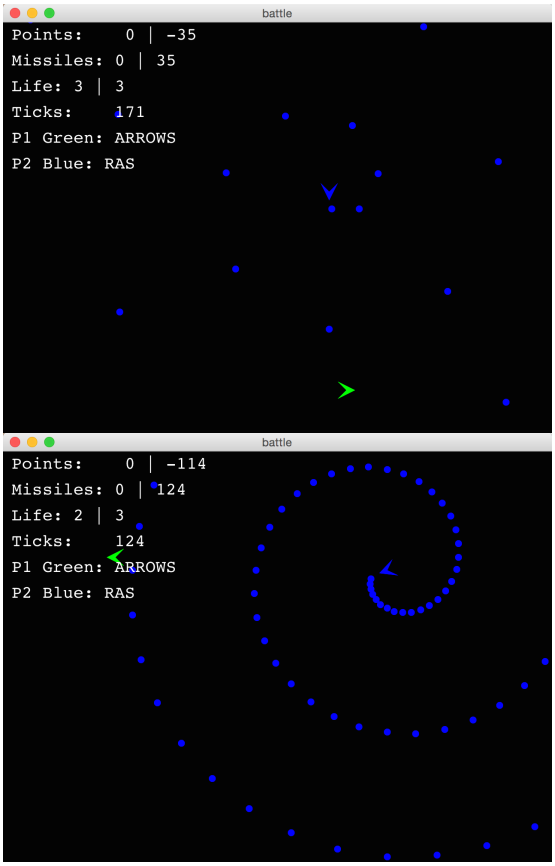

Fig. 1. .Space-battle game with high (left) and low (right) missile cooldown time while fixing the other game parameters. It is more difficult to approach to RAS in the latter case.

A. Random Mutation Hill-Climber

The Random Mutation Hill-Climber (RMHC) is a simple but efficient derivative-free optimisation method mostly used in discrete domains [18], [19]. The pseudo-code of RMHC is given in Algorithm 1. At each generation, an offspring is generated based on the only best-so-far genome (parent) by mutating exactly one uniformly randomly chosen gene. The best-so-far genome is updated if the offspring's fitness value is better or equivalent to the best-so-far.

B. Multi-Armed Bandit Random Mutation Hill-Climber

Multi-Armed Bandit Random Mutation Hill-Climber (MABRMHC), derived from the 2-armed bandit-based RMHC [20], [21], uses both UCB-style selection and mutation rules. MABRMHC selects the coordinate (bandit) with the maximal $urgency$ (Equation 2) to mutate, then mutates the parameter in dimension d to the value (arm) which leads to the maximal reward (Equation 3).

Algorithm 1 Random Mutation Hill-Climber (RMHC).

Require: $\mathcal{X}$: search space
Require: $D = |\mathcal{X}|$: problem dimension (genome length)
Require: $f : \mathcal{X} \mapsto [0, 1]$: fitness function
Randomly initialise a genome $\mathbf{x} \in \mathcal{X}$
$bestFitSoFar \leftarrow 0$
$M \leftarrow 0$ ▷ Counter for the latest best-so-far genome
$N \leftarrow 0$ ▷ Total evaluation count so far
while time not elapsed **do**
  Uniformly randomly select $d \in \{1, \ldots, D\}$
  $\mathbf{y} \leftarrow$ new genome by uniformly randomly mutating the d^{th} gene of $\mathbf{x}$
  $Fit_{\mathbf{x}} \leftarrow fitness(\mathbf{x})$
  $Fit_{\mathbf{y}} \leftarrow fitness(\mathbf{y})$
  $averageFitness_{\mathbf{x}} \leftarrow \frac{bestFitSoFar \times M + Fit_{\mathbf{x}}}{M+1}$
  $N \leftarrow N + 2$ ▷ Update evaluation count
  if $Fit_{\mathbf{y}} \geq averageFitness_{\mathbf{x}}$ **then**
    $\mathbf{x} \leftarrow \mathbf{y}$ ▷ Replace the best-so-far genome
    $bestFitSoFar \leftarrow Fit_{\mathbf{y}}$
    $M \leftarrow 1$
  else
    $bestFitSoFar \leftarrow averageFitness_{\mathbf{x}}$
    $M \leftarrow M + 1$
  end if
end while
return $\mathbf{x}$

For any multi-armed bandit $d \in \{1, 2, \ldots, D\}$, its $urgency_d$ is defined as

$$urgency_d = \min_{1 \leq j \leq Dim(d)} \left(\Delta_d(j) + \sqrt{\frac{2 \log(\sum_{k=1}^{Dim(d)} N_d(k))}{N_d}} + \omega \right), \quad (2)$$

where $N_d(k)$ is the number of times the k^{th} value is selected when the d^{th} coordinate is selected; N_d is the number of times the d^{th} coordinate is selected to mutate, thus $N_d = \sum_{k=1}^{Dim(d)} N_d(k)$; $\Delta_d(k)$ is the maximal difference between the fitness values if the value k is mutated to when the d^{th} dimension is selected, i.e., the changing of fitness value; ω denotes a uniformly distributed value between 0 and $1e^{-6}$ which is used to randomly break ties. Once the coordinate to mutate (eg. d^*) is selected, the index of the value to mutated to is determined by

$$k^* = \underset{1 \leq k \leq Dim(d^*)}{\operatorname{argmax}} \left(\bar{\Delta}_{d^*}(k) + \sqrt{\frac{2 \log(N_{d^*})}{N_{d^*}(k)}} + \omega \right), \quad (3)$$

where $\bar{\Delta}_{d^*}(k)$ denotes the average changing of fitness value if the value k is mutated to when the dimension d^* is selected.

The pseudo-code of MABRMHC is given in Algorithm 2.

In this work, we model each of the game parameter to optimise as a bandit, and the legal values for the parameter as the arms of this bandit. The search space is folded in the sense that it takes far less computational cost to mutate and

Algorithm 2 Multi-Armed Bandit Random Mutation Hill-Climber (MABRMHC). $Dim(d)$ returns the number of possible values in dimension d. ω denotes a uniformly distributed value between 0 and $1e^{-6}$ which is used to randomly break ties.

Require: $\mathcal{X}$: search space
Require: $D = |\mathcal{X}|$: problem dimension (genome length)
Require: $f : \mathcal{X} \mapsto [0, 1]$: fitness function
Randomly initialise a genome $\mathbf{x} \in \mathcal{X}$
$bestFitSoFar \leftarrow 0$
$M \leftarrow 0$ ▷ Counter for the latest best-so-far genome
$N \leftarrow 0$ ▷ Total evaluation count so far
for $d \in \{1, \ldots, D\}$ **do**
  $N_d = 0$
  for $k \in \{1, \ldots, Dim(d)\}$ **do**
    $N_d(k) = 0$, $\Delta_d(k) = 0$, $\bar{\Delta}_d(k) = 0$
  end for
end for
while time not elapsed **do**
  $d^* = \underset{1 \leq d \leq D}{\operatorname{argmax}} \left(\min_{1 \leq j \leq Dim(d)} \Delta_d(j) + \sqrt{\frac{2 \log(\sum_{k=1}^{Dim(d)} N_d(k))}{N_d}} + \omega \right)$ ▷ Select the coordinate to mutate (Equation 2)
  $k^* = \underset{1 \leq k \leq Dim(d^*)}{\operatorname{argmax}} \left(\bar{\Delta}_{d^*}(k) + \sqrt{\frac{2 \log(N_{d^*})}{N_{d^*}(k)}} + \omega \right)$ ▷ Select the index of value to take (Equation 3)
  $\mathbf{y} \leftarrow$ after mutating the element d^* of $\mathbf{x}$ to the k^* legal value
  $Fit_{\mathbf{x}} \leftarrow fitness(\mathbf{x})$
  $Fit_{\mathbf{y}} \leftarrow fitness(\mathbf{y})$
  $averageFitness \leftarrow \frac{bestFitSoFar \times M + Fit_{\mathbf{x}}}{M+1}$
  $N \leftarrow N + 2$ ▷ Update the counter
  $\Delta = Fit_{\mathbf{y}} - averageFitness$
  Update $\Delta_{d^*}(k^*)$ and $\bar{\Delta}_{d^*}(k^*)$ ▷ Update the statistic
  $N_d(k) \leftarrow N_d(k) + 1$, $N_d \leftarrow N_d + 1$ ▷ Update the counters
  if $\Delta \geq 0$ **then**
    $\mathbf{x} \leftarrow \mathbf{y}$ ▷ Replace the best-so-far genome
    $bestFitSoFar \leftarrow Fit_{\mathbf{y}}$
    $M \leftarrow 1$
  else
    $bestFitSoFar \leftarrow averageFitness$
    $M \leftarrow M + 1$
  end if
end while
return $\mathbf{x}$

evaluate every legal value of each parameter once than to evaluate mutate and evaluate every legal game instance once.

V. Experimental results

We firstly use the sample agent using a two-player Open-Loop Monte-Carlo Tree Search algorithm provided by the GVG-AI framework, which uses the difference of scores (Eq. 1) of both players as its heuristic (denoted as OLMCTS), as player 1. No modification or tuning has been performed on this sample agent. We implement a consistently rotate-and-shoot agent (denoted as RAS) as the player 2. More precisely, the RAS is a deterministic agent and, by Eq. 1, the OLMCTS aims at maximising $(100 \times nb_k(1) - c \times nb_m(1)) - (100 \times nb_k(2) - c \times nb_m(2))$, where $nb_k(1)$ and $nb_k(2)$ are the numbers of lives subtracted from the RAS and OLMCTS, respectively; $nb_m(1)$ and $nb_m(2)$ indicates the number of launched missiles by OLMCTS and RAS, respectively. Again, this heuristic is already defined in the sample agent, not by us. Basically, a human player could probably choose to play the game in a passive way by avoiding the missiles and not firing at all, and finally win the game.

The landscape of winning rate of OLMCTS against RAS is studied in Section V-A. Section V-B presents the performance of RMHC and MABRMHC with different resampling numbers to generate games in the parameter space detailed previously (Section III-0e) and Section V-C presents their performances in a 5 times larger parameter space.

A. Winning rate distribution

We use a OLMCTS agent as the player 1 and a RAS agent as the player 2. At each game tick, $10ms$ is allocated to each of the agents to decide an action. The average number of iterations performed by OLMCTS is 350. The time to return an action for RAS is negligible.

The average winning rates over 11 and 69 repeated trials of all the 14,400 legal game instances played by OLMCTS against RAS are shown in Fig. 2. The winning rate over 69 trials of each games instance varies between 20% and 100%. Among all the legal game instances, the OLMCTS does not achieve a 100% winning rate in more than 5,000 games.

Fig. 3 demonstrates how the winning rate varies along with the changing of each parameter. The maximal ship speed and the thrust speed have negligible influence on the OLMCTS's average winning rate. Higher the maximal missile speed is or shorter the cooldown time is, higher the average winning rate is. But still, the average winning rate remains above 87%. The most important factor is the cost of firing a missile. It is not surprising, since the RAS fires successively missiles and the number of missiles it fires during each game is constant depending on the cooldown time. the OLMCTS only fires while necessary or it is likely to slash its opponent.

B. Evolving games by RMHC and MABRMHC using different resampling numbers

We use the same agents as described in Section V-A. RMHC (Algorithm 1) and MABRMHC (Algorithm 2) are applied to

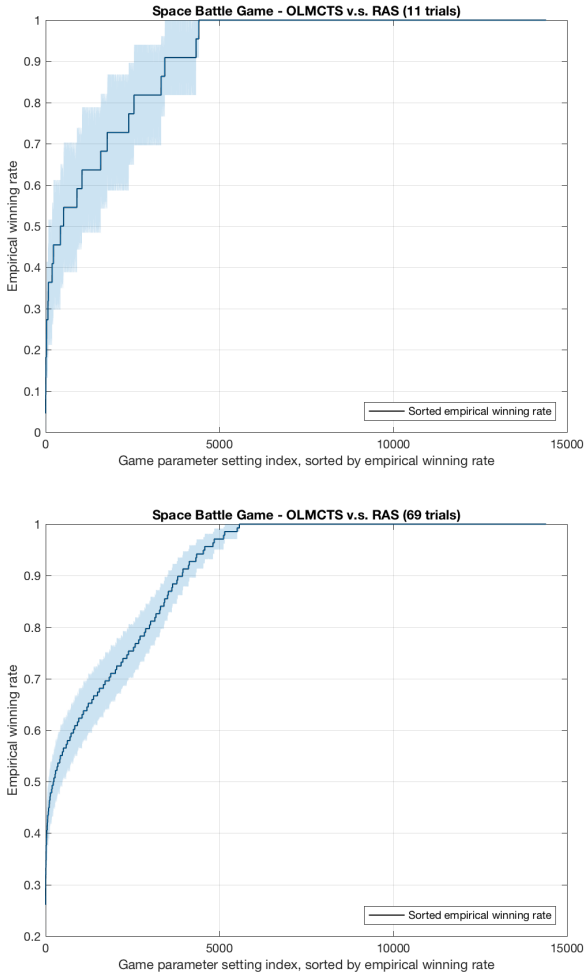

Fig. 2. Empirical winning rates for OLMCTS sorted in increasing order, over 11 trials (left) and 69 trials (right), of all the 14,400 legal game instances played by OLMCTS against RAS. The standard error is shown by the shaded boundary.

optimise the parameters of the space-battle game, aiming at maximising the win probability for the OLMCTS against the RAS. Since the true win probability is unknown, we need to define the fitness of a game using some winning rate by repeating the same game several times, i.e., resampling the game. We define the fitness value of a game g as the winning rate over r repeated games, i.e.,

$$fitness(g) = \frac{1}{r}\sum_{i=1}^{r} GameValue(g). \tag{4}$$

The value of game g is defined as

$$GameValue(g) = \begin{cases} 1, & \text{if OLMCTS wins} \\ 0, & \text{if RAS wins} \\ 0.5, & \text{otherwise (a draw).} \end{cases}$$

A call to $fitness(\cdot)$ is actually based on independent r realizations of the same game. Due to the internal stochastic effects in the game, each realization may return a different

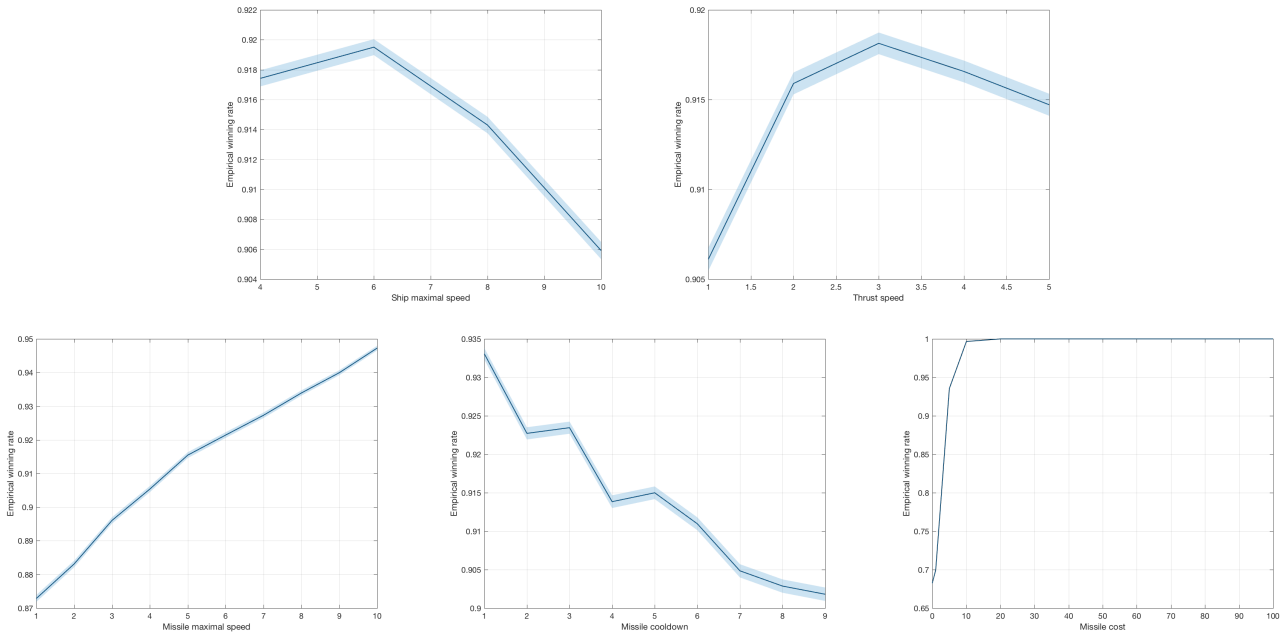

Fig. 3. Empirical winning rates over 69 trials of all the 14,400 legal game instances played by OLMCTS against RAS, classified by the maximal ship speed, the thrust speed, the maximal missile speed, the cooldown time and the cost of firing a missile, respectively. The standard error is shown by the shaded boundary.

game value. We aim at maximising the fitness f in this work. The empirical winning rates shown in Fig. 2 are two example $fitness(\cdot)$ with $r = 11$ (left) and $r = 69$ (right). The strength of noise decreases while repeating the same game more times, i.e., increasing r.

A recent work applied the RMHC and a two-armed bandit-based RMHC with resamplings to a noisy variant of the One-Max problem, and showed both theoretically and practically the importance of choosing a suitable resampling number to accelerate the convergence to the optimum [22], [20], [21]. As the space-battle game introduced previously is stochastic and the agents can be stochastic as well, it is not trivial to model the noise or provide mathematically any optimal resampling number. Therefore, in this work, some resampling numbers are arbitrarily chosen and compared to give a primary idea about the necessary number of resamplings.

Figs. 5 and 4 illustrate the overall performance of RMHC and MABRMHC using different resampling numbers over 1,000 optimisation trials with random starting parameters. A number of 5,000 game evaluations is allocated as optimisation budget in each trial. In other words, given a resampling number r, the $fitness(\cdot)$ (Eq. 4) is called at most $5,000/r$ times.

RMHC and MABRMHC using smaller resampling number achieve a faster move towards to the neighborhood of the optimum at the beginning of optimisation, however, they do not converge to the optimum along with time; despite the slow speed at the beginning, RMHC and MABRMHC using larger resampling number finally succeed in converging to the optimum in the limited budget. A dynamic resampling number which *smoothly* increases with the number of generations will be favourable.

Using smaller budget, MABRMHC reaches the neighborhood of the optimum faster than RMHC. While the current best-so-far fitness is near the optimal fitness value, it's not surprising to see the jagged curves (Fig. 4, right) while the game evaluation consumed is moderate. The drop to the valley dues to the exploration of MABRMHC, then it manages to return to the previous optimum found or possibly find another optimum. Along with the increment of budget, i.e., game evaluations, the quality of best-so-far games found by MABRMHC remains stable.

C. Evolving games in a larger search space

All the 5 parameters considered previously are used for evolving the game rules. In this section, we expand the parameter space by taking into account a parameter for graphical object: the radius of ship. The legal values for ship's radius are $10, 20, 30, 40$ and 50. Thus, the search space is 6-dimensional and the total number of possible games is increase to 72,000 (5 times larger).

Instead of an intelligent agent and a deterministic agent, we play the same OLMCTS agent (with 350 iterations), which has been used previously in Section V-A and Section V-B, against two of its instances: a OLMCTS with 700 iterations and a OLMCTS with 175 iterations, denoted as OLMCTS700 and OLMCTS175 respectively. The same optimisation process using RMHC and MABRMHC is repeated separately, using 1,000 game evaluation. The resampling numbers used are 5 and 50, the ones which have achieved either fastest convergence at the beginning or provides the best recommendation

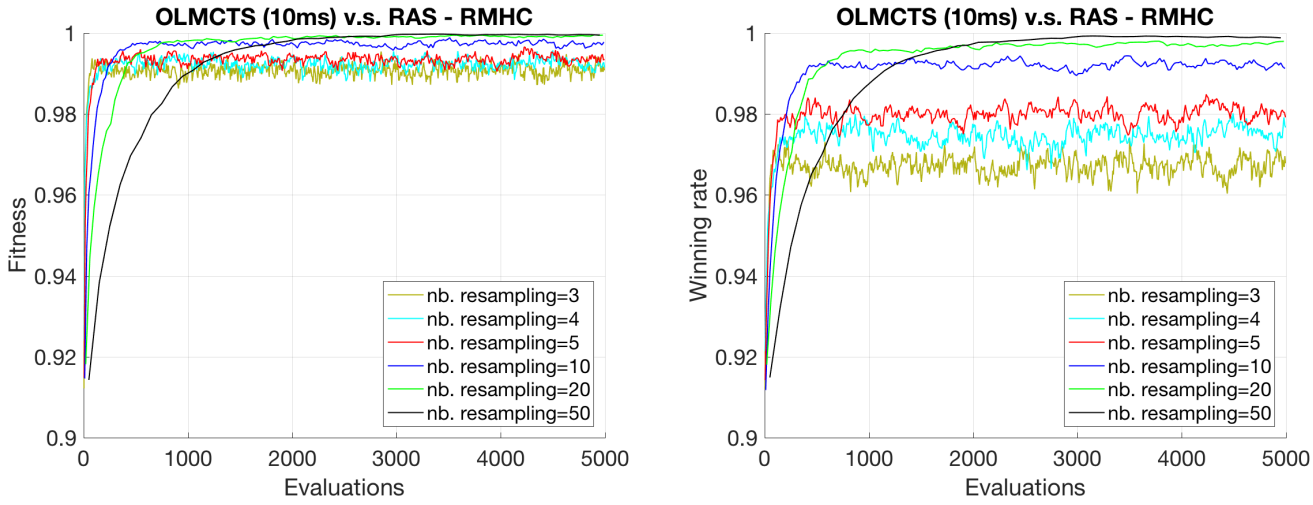

Fig. 4. Average fitness value (left) with respect to the evaluation number over 1,000 optimisation trials by RMHC. The average winning rate of recommended game instances at each generation are shown on the right. The standard error is shown by the shaded boundary.

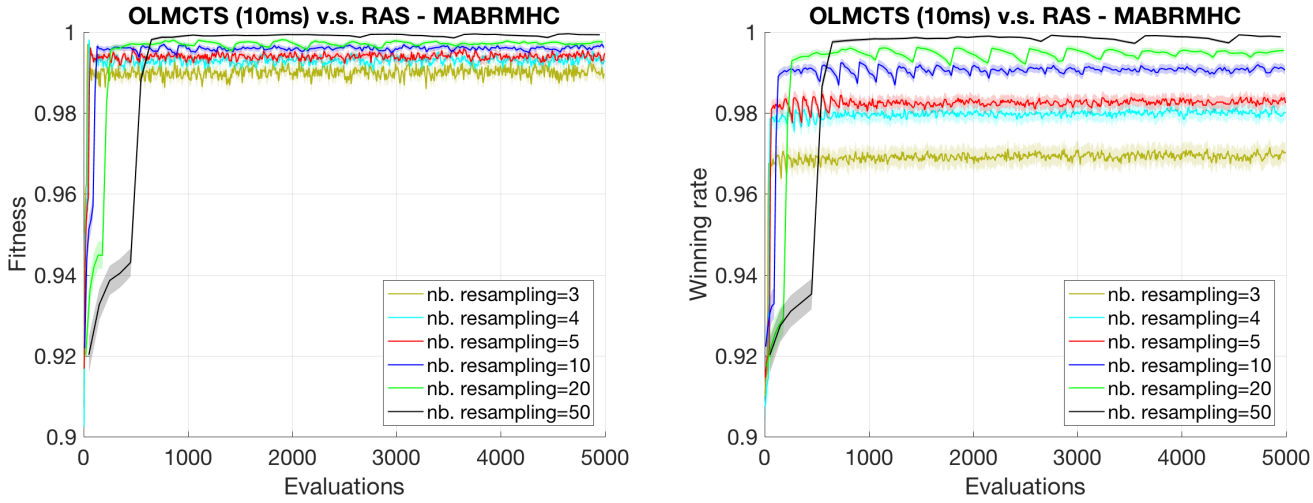

Fig. 5. Average fitness value (left) respected to the evaluation number over 1,000 optimisation trials by MABRMHC. The average winning rate of recommended game instances at each generation are shown on the right. The standard error is shown by the shaded boundary.

at the end of optimisation (after 1,000 game evaluations), respectively. We aim at verifying if the same algorithms still perform well in a larger parameter space and with smaller optimisation budget.

Fig. 6 shows the average fitness value respected to the number of game evaluations over 11 optimisation trials. Resampling 50 times (black curves in Fig. 6) the same game instance guarantee a more accurate winning rate, while resampling 5 times (red curves in Fig. 6) seems to converge faster.

To validate the quality of recommendations, we play each recommended game instance, optimised by playing OLMCTS against OLMCTS175, 100 times using the OLMCTS175 and a random agent, which uniformly randomly returns a legal action. The idea is to verify that the game instances optimised for OLMCTS, are still playable and beneficial for OLMCTS instance with small number of iterations. The statistic is summarised in Table II. The game instances recommended by RMHC and MABRMHC after optimising for OLMCTS with more iterations are still beneficial for the OLMCTS with less iterations. The game is still very difficult for the random agent.

TABLE II
AVERAGE WINNING RATE (%) OVER 11 RECOMMENDATIONS AFTER OPTIMISATION USING 1,000 GAME EVALUATIONS, WITH DIFFERENT RESAMPLING NUMBERS. EACH GAME HAS BEEN REPEATED 100 TIMES.

Algorithm	5 samples	50 samples
RMHC	86.00	91.8182
MABRMHC	81.23	80.9545

D. But what are the evolved games actually like?

To understand the results of the optimisation process, we visually inspected a random sample of games that had been found to have high fitness in the optimisation process, and compared these with several games that had low fitness.

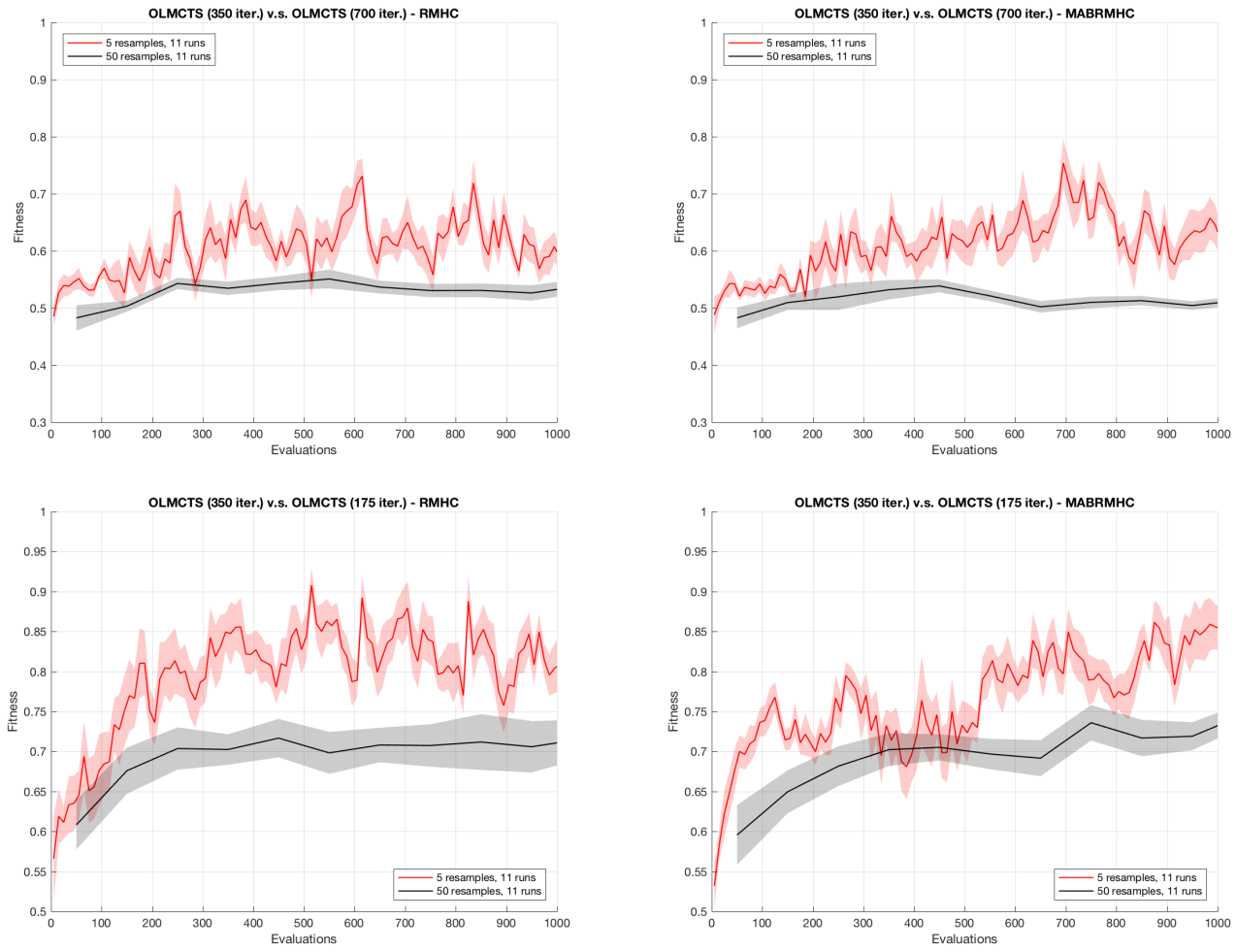

Fig. 6. Average fitness value respected to the evaluation number over 11 optimisation trials by RMHC (left) and MABRMHC (right) using different resampling numbers. The games are played by OLMCTS with 350 iterations against OLMCTS with 700 iterations (top) or OLMCTS against OLMCTS with 175 iterations (bottom). The standard error is shown by the shaded boundary.

We can discern some patterns in the high-fitness games. One of them is to simply have a very high cost for firing missiles. This is somewhat disappointing, as it means that the OLMCTS agent will score higher simply by staying far away from the RAS agent. The latter will quickly reach large negative scores.

A more interesting pattern was to have low missile costs, slow missiles, fast turning speed and fast thrusters. This resulted in a behaviour where the OLMCTS agent coasts around the screen in a mostly straight line, most of the time out of reach of the RAS agent's missiles. When it gets close to the RAS agent, the OLMCTS turns to intercept and salvo of missiles (which typically all hit), and then flies past.

In contrast, several of the low-fitness games have low missile costs and low cool-down times, so that the RAS agent effectively surrounds itself with a wall of missiles. The OLMCTS agent will occasionally attack, but typically loses more score from getting hit and than it gains from hitting the RAS agent. An example of this can be seen in Fig. 3.

It appears from this that the high-fitness games, at least those that do not have excessive missile costs, are indeed deeper games in that skilful play is possible.

VI. Conclusion and Further Work

The work described in this paper makes several contributions in different directions. Our main aim in this work is to provide an automatic game tuning method using simple but efficient black-box noisy optimisation algorithms, which can serve as a base-level game generator and part on an AI-assisted game design tool, assisting a human game designer with tuning the game for depth. The baseline game generator can also help with suggesting game variants that a human designer can build on. Conversely, instead of initialising the optimisation with randomly generated parameters in the search space (as what we have done in this paper), human game designers can provide a set of possibly good initial parameters with their knowledge and experiences.

The game instance evolving provides a method for automatic game parameter tuning or for automatically designing

new games or levels by defining different fitness function used by optimisation algorithms. Even a simple algorithm such as RMHC may be used to automate game tuning. The application of other optimisation algorithms is straightforward.

The two tested optimisation algorithms achieve fast convergence towards the optimum even with a small resampling number when optimising for the OLMCTS against the RAS. Using dynamic non-adaptive or adaptive resampling numbers increasing with the generation number, such as the resampling rules discussed in [23], to take the strength of both small and big numbers of resamplings will be favourable.

Though the primary application of MABRMHC to the space-battle game shows its strength, there is still more to explore. For instance, the selection between parent and offspring is still achieved by resampling each of them several times and comparing their average noise fitness value. However, the classic bandit algorithm stores the average reward and the times that each sampled candidate has been re-evaluated, which is also a form of resampling. We are not making use of this information while making the choice between the parent and offspring at each generation. Using a better recommendation policy (such as UCB or most visited) seems like a fruitful avenue of future work. Another potential issue is the dependencies between the parameters to be optimised in some games or other real world problems. A N-Tuple Bandit Evolutionary Algorithm [24] is proposed to handle such case.

The study of the winning rate distribution and landscape over game instances helps us understand more about the game difficulty. Another possible future work is the study of fitness distance correlation across parameters. Isaksen et al. [5] used Euclidean distance for measuring distance between game instances of *Flappy Bird* and discovered that such a simple measure can be misleading, since the difference between game instances does not always reflect the difference between their parameter values. We observe the same situation when analysing the landscape of fitness value by the possible values of individual game parameter (Fig. 3).

Though we focus on a discrete domain in this work, it's obviously applicable to optimise game parameters in continuous domains, either by applying continuous black-box noisy optimisation algorithms or by discretising the continuous parameter space to discrete values. Evolving parameters for some other games, such as the games in GVG-AI framework, is another interesting extension of this work.

The approach is currently constrained by the limited intelligence of the GVG-AI agent we used, the proof of which is that on many instances of the game a reasonable human player is able to defeat both rotate-and-shoot (RAS) and the OLMCTS players. This problem will be overcome over time as the set of available GVG-AI agents grows.

REFERENCES

[1] J. Togelius and J. Schmidhuber, "An experiment in automatic game design." in *Proceedings of the 2008 IEEE Conference on Computational Intelligence and Games*, 2008, pp. 111–118.

[2] C. Browne and F. Maire, "Evolutionary game design," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 1, pp. 1–16, 2010.

[3] M. Cook, S. Colton, and A. Pease, "Aesthetic considerations for automated platformer design." in *The Eighth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2012.

[4] N. Shaker, J. Togelius, and M. J. Nelson, "Procedural content generation in games: A textbook and an overview of current research," *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, 2016.

[5] A. Isaksen, D. Gopstein, J. Togelius, and A. Nealen, "Discovering unique game variants," in *Computational Creativity and Games Workshop at the 2015 International Conference on Computational Creativity*, 2015.

[6] J. Liu, D. Pérez-Liébana, and S. M. Lucas, "Rolling horizon coevolutionary planning for two-player video games," in *Proceedings of the IEEE Computer Science and Electronic Engineering Conference (CEEC)*, 2016.

[7] B. Pell, "Metagame in symmetric chess-like games," 1992.

[8] J. Togelius, R. De Nardi, and S. M. Lucas, "Towards automatic personalised content creation for racing games," in *2007 IEEE Symposium on Computational Intelligence and Games*. IEEE, 2007, pp. 252–259.

[9] 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.

[10] 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.

[11] N. Sorenson and P. Pasquier, "Towards a generic framework for automated video game level creation," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2010, pp. 131–140.

[12] D. Ashlock, "Automatic generation of game elements via evolution," in *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*. IEEE, 2010, pp. 289–296.

[13] M. Cook and S. Colton, "Multi-faceted evolution of simple arcade games." in *Proceedings of the 2011 IEEE Conference on Computational Intelligence and Games*, 2011, pp. 289–296.

[14] M. J. Nelson and M. Mateas, "Towards automated game design," in *Congress of the Italian Association for Artificial Intelligence*. Springer, 2007, pp. 626–637.

[15] E. J. Powley, S. Gaudl, S. Colton, M. J. Nelson, R. Saunders, and M. Cook, "Automated tweaking of levels for casual creation of mobile games," 2016.

[16] F. Lantz, A. Isaksen, A. Jaffe, A. Nealen, and J. Togelius, "Depth in strategic games," in *under review*, 2017.

[17] T. S. Nielsen, G. A. Barros, J. Togelius, and M. J. Nelson, "General video game evaluation using relative algorithm performance profiles," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2015, pp. 369–380.

[18] S. M. Lucas and T. J. Reynolds, "Learning DFA: Evolution versus Evidence Driven State Merging," in *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, vol. 1. IEEE, 2003, pp. 351–358.

[19] ——, "Learning deterministic finite automata with a smart state labeling evolutionary algorithm," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 27, no. 7, pp. 1063–1074, 2005.

[20] J. Liu, D. Peŕez-Liebana, and S. M. Lucas, "Bandit-based random mutation hill-climbing," *arXiv preprint arXiv:1606.06041*, 2016.

[21] D. P.-L. Jialin Liu and S. M. Lucas, "Bandit-based random mutation hill-climbing," in *Evolutionary Computation, 2017. CEC'17. The 2017 Congress on*. IEEE, 2017.

[22] J. Liu, M. Fairbank, D. Pérez-Liébana, and S. M. Lucas, "Optimal resampling for the noisy onemax problem," *arXiv preprint arXiv:1607.06641*, 2016.

[23] J. Liu, "Portfolio methods in uncertain contexts," Ph.D. dissertation, Université Paris-Saclay, 2015.

[24] K. Kunanusont, R. D. Gaina, J. Liu, D. Perez-Liebana, and S. M. Lucas, "The n-tuple bandit evolutionary algorithm for automatic game improvement," in *Evolutionary Computation, 2017. CEC'17. The 2017 Congress on*. IEEE, 2017.