Self-Adaptive Rolling Horizon Evolutionary Algorithms for General Video Game Playing

Raluca D. Gaina, Diego Perez-Liebana, Simon M. Lucas
Game AI Research Group
Queen Mary University of London, UK
{r.d.gaina, diego.perez, simon.lucas}@qmul.ac.uk

Chiara F. Sironi, Mark H.M. Winands
Game AI & Search Group
Maastricht University, NL
{c.sironi,m.winands}@maastrichtuniversity.nl

***Abstract*—For general video game playing agents, the biggest challenge is adapting to the wide variety of situations they encounter and responding appropriately. Some success was recently achieved by modifying search-control parameters in agents on-line, during one play-through of a game. We propose adapting such methods for Rolling Horizon Evolutionary Algorithms, which have shown high performance in many different environments, and test the effect of on-line adaptation on the agent's win rate. On-line tuned agents are able to achieve results comparable to the state of the art, including first win rates in hard problems, while employing a more general and highly adaptive approach. We additionally include further insight into the algorithm itself, given by statistics gathered during the tuning process and highlight key parameter choices.**

I. Introduction

The aim of General Video Game Playing (GVGP) is to create agents that are able to play many potentially unknown video games without using any pre-coded human knowledge [1]. The most successful GVGP agents are largely using techniques based on Monte-Carlo Tree Search (MCTS) and evolutionary-based approaches, often combined with multiple enhancements [2]. One of the most popular and investigated evolutionary-based approaches for GVGP is the Rolling-Horizon Evolutionary Algorithm (RHEA) [3], which considers sequences of actions as individuals that are evolved over time.

A problem that is common when designing agents able to play a wide variety of (video) games is how to configure them optimally for each game they might play. Usually, certain strategies, enhancements or parameter settings work well in some games, but not in other. Examples of this are available for both MCTS [4] and RHEA agents [5]. A common approach to select agents' configurations in GVGP consists of testing all of them on a set of sample games and selecting the one that performs overall best. However, the selected configuration might still be sub-optimal for some games.

For simulation-based agents, like MCTS, previous research proposed a possible way to address this problem, which consists in a method that tunes search-control parameters on-line, i.e. while playing a single run of a game [6], [7]. For evolutionary-based GVGP agents like the ones based on RHEA, however, no attempt at on-line adaptation has been made so far. Therefore, the aim of this paper is to devise a method that adapts the configuration of the RHEA agent on-line. To do so, the on-line tuning method previously proposed for MCTS is adapted. Different allocation strategies have also been proposed in previous work for on-line parameter tuning of MCTS agents [6], [7]. These strategies decide how to distribute the available samples to evaluate the various parameter value combinations. In this paper, we consider the ones that were most successful for MCTS, which are both based on evolutionary algorithms. The first one is based on a standard Evolutionary Algorithm (EA), while the second one uses the N-Tuple Bandit Evolutionary Algorithm (NTBEA) [8]. Moreover, we consider the Naïve Monte Carlo (NMC) [9] allocation strategy, which for MCTS was shown to be the overall best performing strategy among the ones not based on evolutionary algorithms. Finally, we consider as baseline the Multi-Armed Bandit (MAB) allocation strategy, which ignores the combinatorial structure of the parameter-tuning problem, and the Random (RND) allocation strategy, which does not use any statistics to guide the selection of parameter values.

The contributions of this work are two-fold. First, we propose an adaptation of on-line tuning methods for RHEA and test the performance of the tuned RHEA agents on a variety of different problems, highlighting strengths and weaknesses. Second, we perform an in-depth analysis of the algorithm parameters from the perspective of the statistics gathered by the allocation strategies, with the aim of obtaining more insight into the inner-workings of the algorithm.

II. Background

A. Related Work

Recently, attention to on-line adaptation of game-playing agents has increased, also due to the increased interest in general game playing (GGP). In GGP, agents require different settings for every new game, which might not be known in advance. Thus, the optimal configuration has to be learned on-line. One of the first attempts at adapting GGP agents on-line concerned the adaptation of the playing strategy [10]. An on-line mechanism decides how to allocate available samples to evaluate a portfolio of strategies for the agent, and find the one that is best suited for the current game. Results on abstract games showed that a strategy based on a MAB that tries to allocate the highest number of samples to the best playing strategy while still exploring other strategies performs best.

978-1-7281-4533-4/20/$31.00 ©2020 IEEE

Similarly, an on-line mechanism was used to find the best parameter configuration for an MCTS agent depending on the game being played [6], [7]. In this approach, the result of each MCTS simulation is used to evaluate the quality of the parameter values that control the simulation. Moreover, statistics collected so far on the performance of parameter values are used to choose which values to evaluate next. This approach has been tested both on classic board games [6] and on arcade-style video games [7]. On board games, it had positive results, especially when the number of tuned parameters is small. On video games, on-line tuning was shown to be harder, nevertheless promising for a few of them.

There are also approaches that are able to adapt the agent to each new game using off-line training. For example, approaches based on Hyper-Heuristics have been used in the General Video Game AI framework (GVGAI [2]) by Mendes et al. [11]. These approaches train the agent off-line to recognise the best strategy for the game at hand from a portfolio. When the agent has to play a new game, it uses the trained mechanism to select the best strategy depending on some of the game's features. Experiments have shown that these approaches are promising and are able to outperform agents based on standard algorithms.

Although on-line adaptation has not been tested yet for RHEA, previous work has evaluated off-line tuning for RHEA parameters in GVGAI, using the N-Tuple Bandit Evolutionary Algorithm (NTBEA) to set values for a wide range of parameters and improving upon previous best results obtained in several games [12]. The paper further analyses the algorithm itself based on the n-tuple statistics stored in NTBEA, a line of work, which we continue in the present study.

B. Multi-Armed Bandit

The MAB problem [13] is characterised by m independent arms, each of which is associated with a reward distribution. When an arm is played, a reward is obtained as a sample of the corresponding distribution. The goal of a sampling strategy for a MAB is to maximise the sum of rewards obtained by successive plays of the arms. Thus, the strategy has to balance exploration of less sampled arms in order to learn their distribution, with exploitation of arms that produced a high reward. A variety of sampling strategies have been proposed. One of the most used is UCB1 [13], which, in each iteration, selects the arm a^* as shown in Equation 1.

$$a^* = \underset{a \in A}{\operatorname{argmax}} \left\{ \bar{q}_a + C \times \sqrt{\frac{\ln n}{n_a}} \right\} \quad (1)$$

Here, A is the set of available arms, $\bar{q}_a$ is the average payoff over all the plays of arm a, n is the total number of samples from arms, n_a is the number of samples from arm a, and C is the constant that controls the balance between exploitation of good arms and exploration of less visited ones.

C. Rolling Horizon Evolutionary Algorithms for GVGP

Rolling Horizon Evolutionary Algorithms (RHEA) [3] are a sub-class of Evolutionary Algorithms which evolve action sequences (or plans) for games. The baseline algorithm used in this paper begins the game by initialising a population of P individuals, where each individual represents an action sequence of length L. These individuals are evaluated by simulating through the action sequence using a forward model (FM; an internal model of the game, which must be provided in order for this algorithm to be applicable). The final game state reached is evaluated using a heuristic, and the state value becomes the fitness of the individual. All agents presented in this paper use the same heuristic function, which aims to maximise the dynamically normalised game score, while valuing wins much higher, and losses much lower.

Then, for each generation, the best E individuals ($E = 1$) are promoted directly to the next generation through elitism; crossover may be applied to create several offspring, and mutation may be applied to modify the offspring. The offspring are evaluated as described above and $P - E$ best individuals (highest fitness) are promoted to the next generation, where the process repeats. After several such iterations, RHEA returns the first action in the best individual as the selected action to play in the game. For any subsequent game ticks other than the first, a shift buffer is applied. Instead of initialising a new population, the previous final population is carried through to the next game tick, the first action of each individual is removed and a new random action is added at the end; the value of all individuals is discounted by 0.99.

Table I
PARAMETER SEARCH SPACE, TOTAL SIZE 270, OR 99 VALID COMBINATIONS IF PARAMETER DEPENDENCY IS TAKEN INTO ACCOUNT.

Idx	Name	Values
0	Genetic Operator	Crossover + Mutation, Mutation Only, Crossover Only
1	Selection Type	Rank, Tournament, Roulette
2	Crossover Type	Uniform, 1-point, 2-point
3	Mutation Type	Uniform, 1-Bit, 3-Bits, Softmax, Diversity
4	Mutation Transducer	False, True

III. APPROACH

In order to implement a self-adaptive RHEA agent, the parameter space has to be defined and the on-line parameter tuning method used for MCTS has to be adapted. Moreover, strategies that decide how to allocate the available samples to evaluate different parameter combinations are necessary.

A. RHEA Parameter Space

Although several modifications and parameters of RHEA have been previously studied [5], [14]–[16], we focus here on those parameters which have most impact in any one iteration of the algorithm - that is, those that impact the generation of offspring, as summarised in Table I.

Genetic operator. This parameter controls which genetic operators are applied: crossover only (offspring are not mutated), mutation only (offspring are obtained by directly mutating each individual in the population), or both (offspring

Algorithm 1 RHEA action selection with on-line parameter tuning.

```
1: procedure GETACTION(T, λ, μ)
   Input: Tuner T, population size λ, elite size μ.
   Output: First action of best individual in the population
2:     pop ← GETCURRENTPOPULATION()
3:     order individuals in pop by decreasing fitness
4:     while time not elapsed do
5:         p⃗ ← T.SELECTPARAMVALUES()
6:         SETPARAMVALUES(p⃗)
7:         offspring ← GENANDEVALOFFSPRING(pop_[0,...,μ])
8:         order individuals in offspring by decreasing fitness
9:         r ← offspring_0.fitness − pop_0.fitness
10:        T.UPDATEVALUESTATS(p⃗, r)
11:        pop_[μ+1,...,λ] ← offspring_[0,...,λ−μ]
12:        order individuals in pop by decreasing fitness
13:    end while
14:    return pop_0.GETFIRSTACTION()
15: end procedure
```

are generated through crossover, and then mutated). The rest of the parameters may only have an effect on the phenotype if this parameter has a particular value.

Selection type. This operator is used to select parents for crossover. Roulette selection chooses individuals based on probabilities directly proportional to their fitness. Rank selection chooses individuals based on probabilities inversely proportional to their rank in the ordered list of individuals (best individuals first). Tournament selection randomly chooses 40% of individuals, then the best out of the random selection.

Crossover type. This operator is used to combine selected individuals and generate offspring. Uniform crossover randomly selects a value for each gene from either of the parents. n-point crossover divides the individual into $n+1$ slices and alternatively selects the corresponding slices from each parent; we use $n = \{1, 2\}$.

Mutation type. This operator is used to modify offspring. Uniform mutation changes each gene of the individual with a probability of $1/L$. n-bit mutation changes n random genes; we use $n = \{1, 3\}$. Softmax mutation uses Equation 2 to bias mutation towards the beginning of the individual, where changes in genes most affect the phenotype. Diversity mutation logs values explored for all genes and chooses to mutate the gene explored the least, to its least visited value.

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{2}$$

Mutation transducer. This parameter is used to decide new values for mutating genes (no effect in diversity mutation). If this flag is off or the gene is first in the individual, the gene will take a random new value. Otherwise, it will take the value of the previous gene in the sequence; this aims to decrease the jitteriness of the agent, by encouraging action repetition.

B. On-line Parameter Tuning for RHEA

The parameters we are aiming to tune in RHEA control how the entire population of individuals is evolved. Therefore, we need to use the entire population to evaluate the quality of a parameter combination. Algorithm 1 shows how on-line parameter tuning is implemented by the RHEA agent when it has to choose an action for a given game state. Note that the procedure assumes that the RHEA population has already been initialised and evaluated once. Until the search budget expires, the procedure repeats the following steps:

1) Select a new combination of parameter values using the tuner $\mathcal{T}$ (line 5).
2) Set the tuned parameters to the selected values (line 6).
3) Using the elite individuals in the current population, generate new offspring, evaluate (line 7) and order them by decreasing fitness (line 8).
4) Compute the payoff r for the selected combination of parameter values as the difference between the fitness of the best individual in the offspring and the fitness of the best individual in the previous generation (line 9).
5) Update the statistics of the selected parameter combination using the computed payoff (line 10).
6) Update the population by replacing the worst $\lambda - \mu$ individuals with the generated offspring and order it by decreasing fitness (lines 11 and 12).

When the budget expires, the first action of the best individual in the population is returned (line 14).

Note that, to compute the payoff of a parameter combination, we only consider the fitness of the individuals in the offspring and not of the individuals in the entire new population. This is because the parameter values set for an iteration of RHEA only influence how such offspring is generated. Moreover, to compute the payoff of a combination of parameter values we consider only the best individual in the offspring and in the population, because we are interested in finding the parameters that can generate the best possible individual, even if the rest of the population has a low fitness. The action that will be played in the real game is taken from such individual, therefore it will be the only individual that will influence the real game.

C. Allocation Strategies

The parameters of a game-playing agent can be seen as a vector. Therefore, the problem of tuning these parameters for each new game consists in searching the optimal vector of parameters values in a combinatorial search space. Previous work [7] defined this problem as a Combinatorial Multi-Armed Bandit, characterised by the following three components:

- Vector of d variables, $\vec{P} = \{P_1, ..., P_d\}$, where each variable P_i can take m_i different values $V_i = \{v_i^1, ..., v_i^{m_i}\}$.
- Reward distribution $R : V_1 \times ... \times V_d \leftarrow \mathbb{R}$ that depends on the combination of values assigned to the variables.
- Function $L : V_1 \times ... \times V_d \rightarrow \{true, false\}$ that determines which combinations of values are legal.

There are various allocation strategies that decide which combinations of parameter values should be evaluated, how many times and in which order [6], [7]. The ones considered in this paper are described below.

1) Multi-Armed Bandit (MAB): A straightforward solution to deal with a combinatorial multi-armed bandit problem is to translate it to a Multi-Armed Bandit (MAB) [13]. Each arm of the bandit corresponds to a possible legal combination of values for the parameters. Therefore, selecting the next combination of parameter values to evaluate corresponds to choosing one arm of the bandit. In this paper, UCB1 is used as sampling strategy for the bandit, with exploration constant $C_{\text{MAB}} = 0.7$ (this value is taken from previous work [17]). Note that, differently from other allocation strategies used in this paper, the MAB allocation strategy ignores the information on the combinatorial structure of the parameter values. This strategy does not exploit the fact that often a value that is good (or bad) for a parameter in a certain combination of values, is also good (or bad) in general or in many other combinations.

2) Naïve Monte-Carlo (NMC): First proposed to play real-time strategy games [9], it was later applied as an allocation strategy to tune MCTS parameters on-line [6], [7]. The NMC allocation strategy is based on the naïve assumption that the reward associated to a combination of d parameter values, $\vec{p} = \langle p_1, ..., p_d \rangle$, can be approximated by a linear combination of the rewards associated to single parameter values p_i ($i \in \{1, ..., d\}$). This means $R(\vec{p}) \approx \sum_{i=1}^{d} R_i(p_i)$.

When choosing a combination of parameter values, NMC alternates between *exploration* and *exploitation*. In a given iteration, NMC performs exploration with probability ϵ_0 and exploitation with probability $(1 - \epsilon_0)$. During exploration, for each parameter being tuned NMC considers a local multi-armed bandit problem, where each arm corresponds to a possible value for the parameter. A value for each parameter is selected independently, using the UCB1 sampling strategy with exploration constant C_{L}. During exploitation, NMC considers a global multi-armed bandit, where each arm corresponds to a possible combination of parameter values; UCB1 with exploration constant C_{G} is used to select a combination. At the start of the NMC execution, the global bandit has no arms. A new arm is added every time a new combination of parameter values is generated during exploration with the local bandits. After evaluating a selected combination of parameter values, NMC uses the obtained payoff to update statistics in both the global and local bandits. In this paper, the settings for this strategy are the same used in GVGP to tune MCTS [6]: $\epsilon_0 = 0.75$, $C_{\text{L}} = 0.7$, $C_{\text{G}} = 0.7$.

3) Evolutionary Algorithm (EA): Evolutionary Algorithms (EAs) [18] are optimisation algorithms that search for an optimal solution by evolving a population of candidate solutions. As shown in [6], [7], an EA with λ_{EA} individuals and μ_{EA} elites can be used as allocation strategy for the parameter-tuning problem. A combination of parameter values can be seen as an individual, and each single parameter a gene. The EA allocation strategy starts with a randomly generated population of parameter combinations and then evolves it multiple times until the computational budget expires.

During each iteration, EA first evaluates each combination in the population as shown in Algorithm 1, i.e. using it to control the generation of the new population in one iteration of the RHEA algorithm. This means that the fitness of a combination of parameters corresponds to the payoff computed at line 9 in Algorithm 1. Subsequently, EA keeps the μ_{EA} parameter combinations with the highest fitness in the current population (the elite) and uses them to generate the remaining $\lambda_{\text{EA}} - \mu_{\text{EA}}$ new combinations. Each new combination is generated with probability p_{cross} by uniform random crossover between two randomly selected elite individuals, and with probability $(1 - p_{cross})$ by uniformly mutating one bit of a randomly selected elite individual. In this paper, the settings for this strategy are the same used in GVGP to tune MCTS [6]: $\lambda_{\text{EA}} = 50$, $\mu_{\text{EA}} = 25$, $p_{cross} = 0.5$.

4) N-Tuple Bandit Evolutionary Algorithm (NTBEA): First proposed for game parameter tuning [19], NTBEA has also been successfully used for off-line optimisation of a game-playing RHEA agent [8] and has been applied to on-line parameter tuning for MCTS [6], [7].

Like the EA allocation strategy, NTBEA also considers each combination of parameter values as an individual and each single parameter as a gene. In addition, NTBEA uses an *N-Tuple fitness landscape model* to memorise statistics about the parameters. The implementation of this model, similarly to the NMC approach, keeps a local multi-armed bandit for each parameter and a global bandit for the combination of all the parameters. The landscape model can be used to quickly evaluate a parameter combination by computing its average UCB1 value over all the bandits. This quick evaluation is used by the evolutionary algorithm to speed up the evolutionary process, while balancing exploration and exploitation of the various parameter combinations.

More precisely, the NTBEA algorithm starts with a randomly generated combination of parameter values. During each iteration, the current combination of parameter values is used to control an iteration of RHEA. The obtained payoff is used to update the statistics in the fitness landscape model. At this point, x neighbours of the evaluated combination are generated, each by mutating the value of a randomly selected parameter in the combination. The x neighbours are evaluated using the fitness landscape model and the one with the highest average UCB1 value becomes the new considered combination. This process is repeated until the computational budget expires. The settings used in this paper are: $x = 5$, $C_{\text{NTBEA}} = 0.7$ (exploration constant used for UCB1 values).

5) Random: The random allocation strategy has already been evaluated for on-line parameter adaptation both in abstract games [20] and video games [17]. Despite its simplicity, parameter randomisation has been shown to be beneficial in some games, especially when the fixed parameter settings are not optimal, or when time settings are short. The random allocation strategy selects parameter combinations randomly among all the feasible combinations of parameter values. This means that each iteration of the RHEA algorithm, and thus the generation of each new population, is controlled by a randomly selected combination of parameter values. This also means that no statistics collected about the quality of previously tested parameter values or parameter combinations are exploited.

IV. Experimental Setup

We test the performance of each allocation strategy in 20 games from the General Video Game AI framework (GVGAI [2]): "Dig Dug", "Lemmings", "Roguelike", "Chopper", "Crossfire", "Chase", "Camel Race", "Escape", "Hungry Birds", "Bait", "Wait for Breakfast", "Survive Zombies", "Modality", "Missile Command", "Plaque Attack", "Seaquest", "Infection", "Aliens", "Butterflies", "Intersection". Each of these games has 5 levels, which vary the positions and presence of sprites, as well as the map size. Details of each game are given in [12].

This subset of games features a wide variety of scoring systems with both sparse and dense rewards, win conditions and interactions with sprites in the games, all of which are unknown to the agent; further, half of the games are stochastic, resulting in a very noisy optimisation problem. Within GVGAI, the agent receives game states in an abstract form (e.g. "there is a sprite at position (3,14)"), as well as the current game score and basic information about the position and movement of the avatar it controls. Agents have $40ms$ to return an action to be played in the game; the full set of actions includes doing nothing, moving left, right, up and down and a special action (resulting in a different effect depending on the game), which can be reduced in some of the games (e.g. "Aliens" does not allow moving up or down) Given the complexity and variety of the environment, high adaptation to different and unknown situations is essential for success, thus we deem GVGAI an appropriate testbed.

We use different configurations for RHEA in this setting, varying budget, population size and individual length. Larger values for population size and individual length allow for less iterations during the agent's thinking time, and therefore less data points for the tuners, but were generally shown to perform better [5]. We set the default budget for the agent to 1000 forward model calls, which is the average non-tuned RHEA can perform in $40ms$; this allows for robust results across different machines. Further experiments halve the budget, or increase the budget by 5 times to observe tuner performance outside of GVGAI bounds and with varying numbers of iterations. If we format tested algorithm names as "{individual length}$-${population size}$-${budget}", we obtain 6 configurations: 5-10-500, 5-10-1000; 5-10-5000; 10-15-500; 10-15-1000 and 10-15-5000. Given that there are a total of 270 possible parameter combinations (see Table I), none of the configurations are able to even sample all points at least once during a game tick, let alone gather accurate statistics on all the points, making sample efficiency key.

Each agent (combining a RHEA configuration and an allocation strategy) is run 20 times on each of the 5 levels of the 20 games, or 100 times per game. We record the final result of each game (win/loss, score and game tick). The results are compared with current state of the art (SotA) in RHEA, i.e. highest win rates obtained by any previously explored configuration in each game; as a result, different games may have different RHEA configurations as SotA. Given the nature of the experiment, with parameters varied in tuned agents at every iteration, we consider these SotA results a very high bar, but a good comparison. It is worth noting that some of the enhancements that led to SotA results (e.g. Monte Carlo rollouts [15]) are not used in any of the tuned agents presented here, in order to increase the number of iterations available.

Further, we log the number of visits and average score for all combinations of parameters (5-tuples), and for individual parameters (1-tuples), at every game tick, for all tuners (even if they do not use statistics, e.g. RND).

We note that tuner statistics are not reset between game ticks. Initial experiments showed performance to be very similar regardless of discount factor used (0.0 and 0.8 experimented with). We speculate that this is due to the game states not varying widely from one game tick to the next and thus the statistics on parameter choices are more generally applicable. The only exception we noted was in the game "Crossfire", where a discount of 0.8 showed an increase in performance; this warrants further investigation for highly dynamic games.

V. Results and Discussion

This section presents and discusses some of the more interesting results obtained. Full results, log summaries and plots are available on Github[1].

A. Win Rate

We first look at the performance of the tuned agents in the 20 GVGAI games. For the purpose of this analysis, we only consider win rate (i.e. the agent's ability to solve the problem). This is summarised for all RHEA configurations and tuner combinations in Table II, with a particular RHEA configuration (10-15-1000) visualised in Figure 1. No agent is able to beat SotA results on all games, but win rates are largely comparable, and there are several which do perform better in some of the games, with some interesting cases standing out.

In the game "Crossfire", 9 of the agents with configuration 5-10 are able to beat SotA by up to 8%, whereas none of the other variants with increased core parameter values are able to perform very well. This is thought to be largely due to the nature of the game, where more accurate statistics over several generations are most beneficial. Opposite to this, the larger RHEA configuration (10-15) is able to beat SotA results in "Seaquest" and "Butterflies" in 4 agent variations each. These two games generally benefit from longer rollouts as they feature delayed rewards ("Seaquest") and an increasingly sparser environment ("Butterflies"). All of these games feature a variety of rewards and changes to the agent's environment, which require the high adaptability offered by on-line tuning.

We can also observe a difference in performance when the budget is varied. Win rates generally increase with higher budgets, with particular improvement observed in the games "Wait for Breakfast" and "Missile Command" when the budget is set to 5000 FM calls. These games require different skills, but both show long-term effects of actions and require precision in

[1] https://github.com/rdgain/ExperimentData/tree/RHEA-Online-Tuning-20

Table II

RESULTS OF ALL TUNERS FOR ALL RHEA CONFIGURATIONS TESTED. SHOWING AVERAGE WIN RATE IN ALL 20 GAMES; AVERAGE DIFFERENCE IN WIN RATE TO RHEA SOTA IN GAMES IN WHICH THE TUNED AGENT IS BETTER (Δ_{BETTER}) AND WORSE (Δ_{WORSE}), WITH NUMBER OF GAMES IN BRACKETS. HIGHEST WIN RATE, HIGHEST Δ_{BETTER} AND LOWEST Δ_{WORSE} ARE HIGHLIGHTED FOR EACH TUNER.

Tuner	RHEA Configuration	Win Rate	Δ_{Better}	Δ_{Worse}	RHEA Configuration	Win Rate	Δ_{Better}	Δ_{Worse}
EA	5-10-500	40.80 (±8.51)	0.00 (0)	12.25 (17)	10-15-500	43.75 (±8.84)	0.00 (0)	8.78 (17)
	5-10-1000	43.15 (±8.76)	0.00 (0)	11.52 (14)	10-15-1000	45.90 (±8.79)	0.00 (0)	**7.09 (15)**
	5-10-5000	44.30 (±8.83)	**3.00 (1)**	10.87 (13)	10-15-5000	**46.05 (±8.99)**	2.00 (1)	7.52 (14)
MAB	5-10-500	43.05 (±8.49)	0.00 (0)	9.60 (17)	10-15-500	43.45 (±8.76)	1.00 (1)	9.77 (16)
	5-10-1000	43.20 (±8.61)	2.00 (1)	10.82 (15)	10-15-1000	45.10 (±8.97)	1.00 (1)	**7.70 (16)**
	5-10-5000	44.80 (±8.84)	0.00 (0)	33.90 (34)	10-15-5000	**46.05 (±9.14)**	**2.34 (2)**	7.71 (14)
NMC	5-10-500	42.20 (±8.57)	0.00 (0)	11.27 (16)	10-15-500	43.90 (±8.74)	0.00 (0)	9.14 (16)
	5-10-1000	42.35 (±8.51)	0.00 (0)	10.43 (17)	10-15-1000	45.75 (±8.99)	2.34 (2)	**7.60 (15)**
	5-10-5000	44.40 (±8.69)	**3.00 (1)**	8.71 (16)	10-15-5000	**45.85 (±8.80)**	1.30 (3)	7.94 (14)
NTBEA	5-10-500	43.05 (±8.53)	2.00 (1)	11.81 (14)	10-15-500	42.45 (±8.72)	0.00 (0)	10.31 (17)
	5-10-1000	43.30 (±8.54)	0.00 (0)	9.89 (16)	10-15-1000	44.60 (±8.79)	0.00 (0)	7.78 (17)
	5-10-5000	45.20 (±8.69)	**8.00 (1)**	8.55 (15)	10-15-5000	**46.30 (±8.86)**	2.00 (1)	**6.27 (16)**
RND	5-10-500	43.10 (±8.48)	1.00 (1)	12.56 (13)	10-15-500	43.35 (±8.81)	0.00 (0)	9.83 (16)
	5-10-1000	44.75 (±8.73)	2.00 (1)	9.38 (14)	10-15-1000	**45.90 (±8.91)**	1.33 (3)	**7.88 (14)**
	5-10-5000	45.15 (±8.89)	2.84 (2)	9.77 (13)	10-15-5000	45.60 (±9.02)	**2.96 (3)**	9.32 (13)

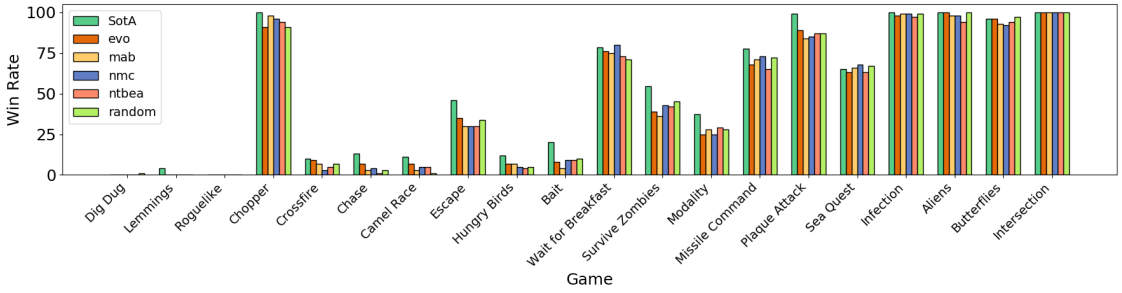

Figure 1. Win rate of all tuners (using RHEA configuration 10-15-1000), compared against RHEA state of the art.

decision making. Thus, the increased budget not only allows for RHEA to find better plans, but also allows the tuners to increase their accuracy in recommending good parameters.

Most interestingly, we remark first wins from some agents in very difficult games where SotA remains at 0%: RND-10-15-1000 wins one game of "Dig Dug", and NMC, NTBEA and RND (with RHEA configuration 10-15-5000) win 2 games of "Roguelike" each (similarly observed in [6]). We consider these clear examples of the benefits of on-line adaptability for tackling seemingly very difficult problems, requiring further investigation.

We see no large differences in overall performance for the tuners, although they do appear to have different strengths. The RND tuner achieves the most better-than-SotA results across all RHEA configurations (3 games for both 10-15-1000 and 10-15-5000), and performs worse in least games as well (13 games for both 5-10-5000 and 10-15-5000). However, NTBEA obtains the highest difference to SotA in winning games (8%), as well as the lowest difference in losing games (6.27%).

B. Parameter Combinations Analysis (5-tuples)

These experiments can also give further insight into what works, and what does not, in the algorithm being tuned. Table III shows the parameter combination chosen most often as the best option, for RHEA 5-10-1000. We further note the Shannon entropy $H(x)$ over all game ticks, which gives an indication of the tuner's consistency. Using this measure, the NMC tuner seems most consistent in its recommendations, while the EA tuner the least; given their win rates (Table II), this could suggest consistency might not be essential to success (e.g. if consistently recommending bad parameter combinations), although there is no strong evidence in this direction.

No tuner seems to agree with others on the best parameter combination for any game; this is likely due to the low number of data points each tuner is able to sample during a game, which do not allow for significant statistics. However, some partial agreements can be observed, which could indicate good combinations of parameters. In particular, we can observe a preference in several games and tuners for the (3, 1) 2-tuple (mutation type, mutation transducer), corresponding to values (softmax, true). There is not a similar case for the crossover parameters (selection type, crossover type), where a wide variety of values are chosen in the best 5-tuples.

Table III

TUPLES CONSIDERED BEST MOST TIMES AND SHANNON ENTROPY $H(x)$ FOR ALL TUNERS, ONE ROW PER GAME, WITH RHEA CONFIGURATION 5-10-1000; SHOWING ONLY THE GENETIC OPERATOR 1-TUPLES. PARAMETER INDEX AND VALUE INDEX CORRESPOND TO ORDER IN TABLE I AND GAME INDEX CORRESPONDS TO LIST IN SECTION IV. MAJORITY AGREEMENT ACROSS TUNERS HIGHLIGHTED FOR 1-TUPLES.

	EA		MAB		NMC		NTBEA		RND	
G	**5-tuple : $H(x)$**	**1-tuple**	**5-tuple : $H(x)$**	**1-tuple**	**5-tuple : $H(x)$**	**1-tuple**	**5-tuple : $H(x)$**	**1-tuple**	**5-tuple : $H(x)$**	**1-tuple**
0	(0, 0, 0, 0, 1) : 5.87	**1**	(1, 0, 0, 0, 1) : 5.37	**1**	(0, 2, 1, 3, 1) : 4.41	2	(2, 2, 2, 4, 1) : 4.81	**1**	(0, 0, 1, 1, 1) : 4.92	2
1	(0, 2, 1, 1, 0) : 6.35	**1**	(1, 2, 2, 3, 0) : 5.24	0	(0, 1, 0, 3, 1) : 4.44	2	(1, 1, 1, 3, 1) : 5.42	**1**	(1, 1, 1, 3, 0) : 4.68	**1**
2	(1, 0, 1, 0, 0) : 5.27	**1**	(0, 0, 2, 3, 1) : 5.25	**1**	(2, 0, 0, 4, 0) : 4.25	2	(0, 2, 0, 5, 0) : 5.32	**1**	(0, 2, 1, 1, 0) : 5.11	**1**
3	(1, 2, 2, 1, 0) : 5.99	**0**	(0, 1, 0, 0, 0) : 5.09	**0**	(2, 2, 1, 5, 1) : 4.21	2	(2, 1, 2, 0, 1) : 5.46	2	(0, 2, 0, 3, 1) : 4.32	**0**
4	(0, 2, 2, 3, 1) : 6.06	**1**	(0, 1, 2, 0, 1) : 4.98	**1**	(1, 1, 1, 3, 1) : 4.28	**1**	(1, 2, 2, 0, 1) : 5.28	**1**	(0, 2, 2, 3, 1) : 4.90	**1**
5	(1, 1, 2, 3, 1) : 6.05	0	(0, 2, 2, 0, 1) : 4.68	**1**	(1, 1, 1, 4, 0) : 3.73	**1**	(0, 1, 1, 0, 0) : 5.69	**1**	(1, 2, 1, 0, 0) : 5.57	2
6	(1, 0, 0, 3, 1) : 5.97	**1**	(1, 0, 0, 1, 0) : 5.21	**1**	(2, 0, 0, 1, 1) : 4.61	0	(1, 0, 1, 1, 0) : 5.27	0	(2, 2, 0, 3, 1) : 5.05	**1**
7	(0, 0, 1, 3, 1) : 6.28	**1**	(0, 2, 0, 3, 1) : 4.73	**1**	(1, 0, 0, 3, 1) : 4.18	2	(0, 0, 2, 3, 1) : 5.72	**1**	(0, 1, 2, 5, 1) : 4.70	0
8	(1, 0, 0, 0, 1) : 6.07	0	(0, 0, 2, 0, 1) : 4.63	**1**	(1, 0, 2, 3, 1) : 4.32	**1**	(1, 0, 0, 4, 1) : 5.70	**1**	(0, 0, 0, 3, 1) : 5.03	2
9	(0, 0, 1, 3, 1) : 6.11	**1**	(2, 0, 1, 3, 0) : 4.97	2	(0, 0, 2, 3, 1) : 4.93	**1**	(0, 2, 0, 4, 1) : 5.76	**1**	(0, 1, 2, 3, 1) : 5.08	2
10	(0, 0, 1, 0, 0) : 5.77	0	(2, 0, 1, 3, 0) : 4.88	2	(0, 1, 2, 4, 1) : 4.20	1	(1, 2, 0, 1, 0) : 5.19	1	(0, 0, 2, 1, 1) : 4.23	2
11	(0, 1, 1, 5, 0) : 6.20	0	(1, 2, 0, 0, 1) : 5.33	**1**	(2, 2, 0, 3, 0) : 4.68	**1**	(2, 0, 2, 0, 1) : 5.51	**1**	(1, 1, 2, 3, 1) : 5.09	0
12	(1, 1, 2, 3, 1) : 6.24	**1**	(0, 1, 1, 3, 1) : 5.02	**1**	(0, 1, 1, 0, 1) : 4.51	2	(1, 0, 2, 3, 1) : 5.50	**1**	(2, 2, 0, 3, 1) : 5.07	**1**
13	(0, 2, 2, 3, 1) : 5.82	**1**	(1, 0, 1, 3, 0) : 4.59	**1**	(1, 0, 2, 0, 0) : 4.11	**1**	(0, 2, 0, 4, 0) : 5.65	**1**	(0, 0, 1, 5, 0) : 5.25	2
14	(0, 0, 0, 3, 0) : 6.10	1	(1, 1, 2, 0, 1) : 4.41	0	(1, 0, 0, 5, 1) : 3.81	1	(0, 1, 0, 3, 1) : 5.19	2	(0, 1, 0, 0, 1) : 5.24	2
15	(1, 1, 1, 1, 1) : 5.95	**2**	(2, 0, 0, 0, 1) : 5.15	0	(1, 1, 1, 4, 0) : 4.46	**2**	(1, 2, 0, 4, 1) : 5.27	1	(0, 2, 0, 3, 1) : 4.92	**2**
16	(1, 2, 1, 0, 1) : 5.42	**1**	(0, 2, 1, 3, 1) : 5.15	**1**	(2, 2, 0, 0, 0) : 4.42	**1**	(1, 1, 0, 4, 1) : 5.49	0	(0, 1, 1, 0, 1) : 5.46	**1**
17	(1, 2, 0, 0, 1) : 5.86	0	(1, 1, 2, 1, 0) : 4.09	2	(1, 1, 0, 1, 0) : 3.83	1	(0, 1, 1, 3, 1) : 5.14	0	(1, 1, 1, 3, 1) : 5.51	2
18	(0, 2, 0, 3, 1) : 5.81	1	(0, 0, 0, 5, 0) : 4.61	2	(1, 1, 1, 5, 0) : 4.21	2	(2, 1, 1, 1, 0) : 5.49	0	(2, 0, 0, 5, 0) : 4.98	1
19	(1, 2, 0, 3, 1) : 6.20	0	(1, 2, 2, 1, 1) : 4.82	**1**	(2, 2, 2, 3, 0) : 4.09	**1**	(2, 0, 2, 3, 0) : 5.33	2	(1, 1, 1, 3, 1) : 5.23	**1**

C. Individual Parameter Analysis (1-tuples)

Lastly, we can analyse each parameter individually and the values each tuner considers best. In Table III, we also highlight the value chosen as the best option for the genetic operator parameter. We can observe more agreement between the different tuners on individual values of parameters, with 12 games showing 3 or more tuners choosing the same value for the genetic operator (mutation only, with two exceptions). In "Chopper", they all show a preference for using both crossover and mutation. This game generally benefits from ample exploration of the action space, which is best obtained with both genetic operators enabled. Whereas in "Seaquest", crossover only is considered best, leading to smaller disturbances, which could be key in a stochastic game with many possible deaths to the agent - a point strengthened by the increase in performance of several tuned agents in this game.

This is similar for most other parameters, with rank selection, 1-bit mutation and mutation transducer enabled being chosen most often; there is no agreement on which crossover type works best in any of the games. All tuners show similar and fairly high levels of consistency (given by Shannon entropy) and variance in their recommendations.

Figure 2 shows that, interestingly, both MAB and NTBEA consider option 3 (softmax) the worst value for mutation in many games, which we previously saw chosen most often in 5-tuples. Although this could be a question of credit assignment, as parameters receive associated values even if they do not impact the phenotype, we consider this an important highlight of the combinatorial problem, as opposed to choosing the best 1-tuple values. In contrast, MAB and NMC do not feature a similar ranking of options for the genetic operator, although they do both favour option 1 (mutation only) in most games.

In Figure 3 we take a look one level deeper into parameter choices per game tick, and how the scores given to each parameter value progresses over the course of a game. The

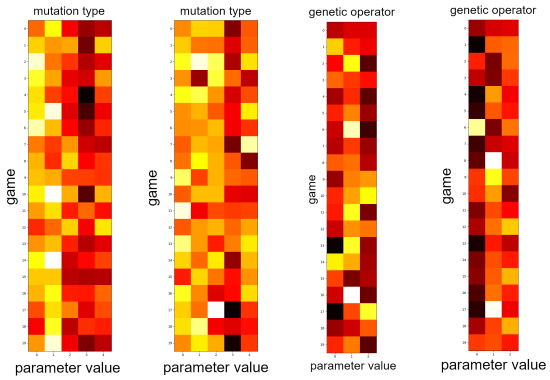

Figure 2. Normalised count of times considered best, per parameter value, one game per row. From left to right: MAB, NTBEA, MAB, NMC. Parameter value index corresponds to list from Table I. All use 5-10-1000 RHEA configuration.

examples included highlight the different approaches of the tuners, with shapes similar across all other parameters for the same tuners as well. MAB seems to always start with overestimations before settling on lower scores, whereas NTBEA and NMC both show upwards trends and are able to make better use of information from previous game ticks to continue improving their parameter recommendations.

VI. CONCLUSIONS

This paper presented the use of various optimisation methods in choosing parameters for the Rolling Horizon Evolutionary Algorithm (RHEA) on-line (i.e. during one playthrough of a game). Five different tuners were used in this context, a standard Evolutionary Algorithm (EA), a Multi-Armed Bandit (MAB), Naive Monte Carlo (NMC), the N-Tuple Bandit Evolutionary Algorithm (NTBEA) and Random (RND). Several budget options, population sizes and individual length were used for RHEA. The tuned agents were tested in

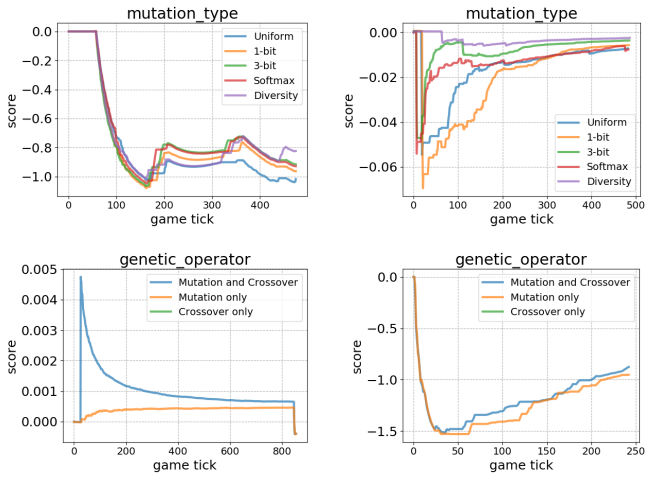

Figure 3. Average parameter values over game ticks. Top row: mutation type, in "Wait for Breakfast" by MAB (left) and NTBEA (right). Bottom row: genetic operator, in "Crossfire" by MAB (left) and NMC (right). All use 5-10-1000 RHEA configuration, in winning game instances.

20 games from the General Video Game AI framework and win rate, as well as parameter combinations and individual parameter choices were analysed in detail.

Win rates of tuned agents were comparable to the high bar set by the RHEA state of the art, surpassing it in several games. We highlight that this approach is more general and highly adaptive, as opposed to hand-picked SotA results. Generally, longer RHEA rollouts and higher budgets led to better outcomes, although the opposite led to specific improvements in the game "Crossfire". Tuner performance was very similar, with RND and NTBEA standing out in a few cases; longer games could lead to more significant results and differences in tuners. A deeper analysis into the strengths and weaknesses of the tuners, and better initialisation methods, is one avenue for future work, as well as exploring different allocation strategies such as Bayesian Optimisation; further, could the tuner itself be tuned, or, the choice of tuner included as a parameter, for a branching hyper-parameter optimisation algorithm?

There did not appear to be a particular combination of parameters, which worked best in all games, or even the same recommendation by all tuners in a game. This emphasises the difficulty of this highly stochastic problem. However, some combinations of parameters did stand out as better than others, such as softmax mutation, used with a mutation transducer. 1-tuple analysis suggested 1-bit mutation to be the best instead when dependencies are ignored, using mutation only as the genetic operator and keeping the mutation transducer enabled.

We also observed interesting behaviour and novel results in very difficult problems such as "Dig Dug" and "Rogue"; obtaining more data points where agents win these games could give important information on strategies for tackling such problems. Similarly, we saw a difference in highly dynamic games when discounting the tuner statistics between game ticks, which is worth further investigation and could boost performance in this class of problems.

Acknowledgments

This work was partly funded by the EPSRC CDT in Intelligent Games and Game Intelligence (IGGI) EP/L015846/1. This work used Queen Mary's Apocrita HPC facility. http://doi.org/10.5281/zenodo.438045

References

[1] J. Levine *et al.*, "General Video Game Playing," in *Artificial and Computational Intelligence in Games*, ser. Dagstuhl Follow-Ups, S. Lucas *et al.*, Eds. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, vol. 6, pp. 77–83.

[2] D. Perez-Liebana *et al.*, "General Video Game AI: A Multitrack Framework for Evaluating Agents, Games, and Content Generation Algorithms," *IEEE Trans. on Games*, vol. 11, no. 3, pp. 195–214, 2019.

[3] D. Perez-Liebana, S. Samothrakis, S. M. Lucas, and P. Rolfshagen, "Rolling Horizon Evolution versus Tree Search for Navigation in Single-Player Real-Time Games," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2013, pp. 351–358.

[4] D. J. N. J. Soemers, C. F. Sironi, T. Schuster, and M. H. M. Winands, "Enhancements for Real-Time Monte-Carlo Tree Search in General Video Game Playing," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2016, pp. 436–443.

[5] R. D. Gaina, J. Liu, S. M. Lucas, and D. Perez-Liebana, "Analysis of Vanilla Rolling Horizon Evolution Parameters in General Video Game Playing," in *Applications of Evolutionary Computation*, ser. LNCS, G. Squillero and K. Sim, Eds., vol. 10199. Springer, 2017, pp. 418–434.

[6] C. F. Sironi *et al.*, "Self-Adaptive MCTS for General Video Game Playing," in *International Conference on the Applications of Evolutionary Computation*. Springer, 2018, pp. 358–375.

[7] C. F. Sironi, J. Liu, and M. H. M. Winands, "Self-adaptive Monte-Carlo Tree Search in General Game Playing," *IEEE Transactions on Games*, 2020, in press.

[8] S. M. Lucas, J. Liu, and D. Perez-Liebana, "The N-Tuple Bandit Evolutionary Algorithm for Game Agent Optimisation," in *Congress on Evolutionary Computation*. IEEE, 2018.

[9] S. Ontañón, "Combinatorial Multi-armed Bandits for Real-Time Strategy Games," *Journal of AI Research*, vol. 58, pp. 665–702, March 2017.

[10] M. Świechowski and J. Mańdziuk, "Self-Adaptation of Playing Strategies in General Game Playing," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, no. 4, pp. 367–381, Dec. 2014.

[11] A. Mendes, J. Togelius, and A. Nealen, "Hyper-Heuristic General Video Game Playing," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2016, pp. 94–101.

[12] R. D. Gaina, S. Devlin, S. M. Lucas, and D. Perez-Liebana, "Rolling Horizon Evolutionary Algorithms for General Video Game Playing," *arxiv:2003.12331*, 2020.

[13] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-Time Analysis of the Multiarmed Bandit Problem," *ML*, vol. 47, no. 2–3, pp. 235–256, 2002.

[14] R. D. Gaina, S. M. Lucas, and D. P. Liébana, "Population Seeding Techniques for Rolling Horizon Evolution in General Video Game Playing," in *Proceedings of the Congress on Evolutionary Computation*, June 2017, pp. 1956–1963.

[15] R. D. Gaina, S. M. Lucas, and D. Perez-Liebana, "Rolling Horizon Evolution Enhancements in General Video Game Playing," in *Conf. on Computational Intelligence and Games*. IEEE, 2017, pp. 88–95.

[16] ——, "Tackling Sparse Rewards in Real-Time Games with Statistical Forward Planning Methods," in *AAAI Conference on Artificial Intelligence (AAAI-19)*, vol. 33, 2019, pp. 1691–1698.

[17] C. F. Sironi and M. H. M. Winands, "Analysis of Self-Adaptive Monte Carlo Tree Search in General Video Game Playing," in *Conference on Computational Intelligence and Games (CIG)*. IEEE, 2018, pp. 397–400.

[18] D. Ashlock, *Evolutionary Computation for Modeling and Optimization*. Springer Science & Business Media, 2006.

[19] K. Kunanusont *et al.*, "The N-tuple Bandit Evolutionary Algorithm for Automatic Game Improvement," in *Congress on Evolutionary Computation*. IEEE, 2017, pp. 2201–2208.

[20] C. F. Sironi and M. H. M. Winands, "Comparing Randomization Strategies for Search-Control Parameters in MCTS," in *Conf. on Games (COG)*. IEEE, 2019.