


Self-Adaptive MCTS for General Video Game Playing

Chiara F. Sironi¹, Jialin Liu², Diego Perez-Liebana², Raluca D. Gaina², Ivan Bravi², Simon M. Lucas², and Mark H. M. Winands¹

¹ Games & AI Group, Department of Data Science and Knowledge Engineering
Maastricht University, Maastricht, The Netherlands

{c.sironi,m.winands}@maastrichtuniversity.nl

² Game AI Group, School of Electronic Engineering and Computer Science
Queen Mary University of London, London, UK

{jialin.liu,diego.perez,r.d.gaina,i.bravi,simon.lucas}@qmul.ac.uk

Abstract. Monte-Carlo Tree Search (MCTS) has shown particular success in General Game Playing (GGP) and General Video Game Playing (GVGP) and many enhancements and variants have been developed. Recently, an on-line adaptive parameter tuning mechanism for MCTS agents has been proposed that almost achieves the same performance as off-line tuning in GGP.

In this paper we apply the same approach to GVGP and use the popular General Video Game AI (GVGAI) framework, in which the time allowed to make a decision is only 40ms. We design three Self-Adaptive MCTS (SA-MCTS) agents that optimize on-line the parameters of a standard non-Self-Adaptive MCTS agent of GVGAI. The three agents select the parameter values using Naïve Monte-Carlo, an Evolutionary Algorithm and an N-Tuple Bandit Evolutionary Algorithm respectively, and are tested on 20 single-player games of GVGAI.

The SA-MCTS agents achieve more robust results on the tested games. With the same time setting, they perform similarly to the baseline standard MCTS agent in the games for which the baseline agent performs well, and significantly improve the win rate in the games for which the baseline agent performs poorly. As validation, we also test the performance of non-Self-Adaptive MCTS instances that use the most sampled parameter settings during the on-line tuning of each of the three SA-MCTS agents for each game. Results show that these parameter settings improve the win rate on the games *Wait for Breakfast* and *Escape* by 4 times and 150 times, respectively.

Keywords: MCTS, on-line tuning, self-adaptive, robust game playing, general video game playing

1 Introduction

Monte-Carlo Tree Search (MCTS) [1,2] is a simulation-based search technique that Yannakakis and Togelius [3] list among the commonly used methods in games. Browne *et al.* [4] reviewed the evolution of MCTS, its variations and their successful applications till 2012. Different techniques have been studied for enhancing the MCTS variants, such as the All Moves As First (AMAF) [5], the Rapid Action Value Estimation (RAVE) [6], Generalized RAVE (GRAVE) [7], HRAVE [8] and the Move Average

Sampling Technique (MAST) [9]. Powley *et al.* introduced the ICARUS (Information Capture And ReUse Strategy) framework for describing and combining such enhancements [10].

Moreover, MCTS has been proved to be a success in many domains, including the Go game [11,12] and General (Video) Game Playing [13,14,15]. General Video Game Playing (GVGP) [14,15] aims at creating agents that are capable of playing any unknown video game successfully without using prior knowledge or intervention by human beings. The General Video Game AI (GVGAI) framework³ has been implemented for this research purpose [14]. Many successful General Video Game Playing agents are MCTS-based, like YOLOBOT, the agent that won the GVGAI Single-Player Planning Championship in 2017, and *MaastCTS2* [16], the agent that won the GVGAI Single-Player Planning Championship in 2016.

One of the difficulties of GVGP and the GVGAI competition is parameter tuning for AI agents. Examples are the exploration factor and the play-out depth of MCTS-based agents, or population size, mutation probability and planning horizon for the rolling horizon evolutionary agents. A common approach is to tune the parameters off-line with some given games. Gaina *et al.* [17] varied the population size and planning horizon of a vanilla Rolling Horizon Evolutionary Algorithm (RHEA) and compared their performance on a subset of games in the GVGAI framework. Bravi *et al.* [18] used Genetic Programming (GP) to evolve game-specific Upper Confidence Bound (UCB) alternatives, each of which outperformed the MCTS using standard UCB1 (Equation 1) on at least one of the tested games. These UCB alternatives can be used to build a portfolio of MCTS agents to achieve robust game playing. However, such off-line tuning is computationally expensive and game dependent. Tuning the parameters of a given agent off-line for a new game is therefore sometimes not possible.

Recently, Sironi and Winands [19] have proposed on-line adaptive parameter tuning for MCTS agents and almost achieved the same performance as off-line tuning in General Game Playing. Their approach is based on the idea of interleaving parameter tuning with MCTS. Before each MCTS simulation a different combination of parameter values is selected to control the search. The reward obtained by the simulation is used to update some statistics on the performance of such combination of parameter values. These statistics are then used to choose which parameter values will control the next simulations. Four allocation strategies are proposed in [19] to decide which parameter values should be evaluated for each MCTS simulation and in which order.

In this work, we apply the same approach for tuning on-line the parameters K (the UCB exploration factor) and D (the depth limit for the search) of a standard MCTS agent, *sampleMCTS*, of the GVGAI framework. First of all we verify if the on-line tuning approach can be applied successfully to GVGP by testing the most promising among the four allocation strategies presented in [19], Naïve Monte-Carlo (NMC). Second, we want to see if the performance on GVGP of the on-line tuning mechanism can be further improved by using the principles of evolutionary algorithms. Therefore, we propose two more allocation strategies, one based on an Evolutionary Algorithm (EA) and one based on an N-tuple Bandit Evolutionary Algorithm (NTupleBanditEA). Finally, to validate the allocation strategies we evaluate the performance of the instances of *sampleMCTS*

³ <http://www.gvgai.net>

that use as fixed parameter settings the combinations of values that were used the most during game playing by each of the proposed allocation strategies.

This paper is structured as follows. Section 2 introduces the background, including GVGAI, MCTS and the on-line parameter tuning problem. Section 3 describes the approach and tuning algorithms. Section 4 presents the design of experiments and 5 analyzes the results. Finally, Section 6 concludes and proposes some future research.

2 Background

This section briefly introduces the General Video Game AI framework (Subsection 2.1), the principles of Monte-Carlo Tree Search (Subsection 2.2) and the formulation of the on-line parameter tuning problem (Subsection 2.3).

2.1 General Video Game AI

The General Video Game AI (GVGAI) framework was initially designed and developed by the Games Intelligence Group at the University of Essex (UK) aiming at using it as a research and competition framework for studying General Video Game Playing (GVGP). The GVGAI consists of five tracks: two planning tracks (single- and two-player) where the forward model of every game is available [14,15]; the single-playing learning track where no forward model is given [20]; the level generation track [21] and rule generation track [22]. The games were defined in Video Game Description Language (VGDL) [23] and the framework was mainly written in Java. More about the tracks and competitions can be found on the GVGAI website. In this paper, we focus on a subset of the GVGAI single-player games. More about the game set is presented in Subsection 4.1. Compared to the Atari Learning Environment (ALE) [24] framework, GVGAI has the advantage of being more extensible, meaning that it is much easier to add new games and variations of those games, and also offers two-player games which provide a greater range of challenges than single player games. ALE currently has the advantage of offering commercial games, albeit from a few decades ago.

2.2 Monte-Carlo Tree Search

MCTS is a best-first search algorithm that incrementally builds a tree representation of the search space of a problem (e.g., a game) and estimates the values of states by performing simulations [1,2]. An iteration of the MCTS consists of four phases: (i) **Selection:** starting from the root node of the tree a *selection strategy* is used to select the next actions to visit until a node is reached that is not fully expanded, i.e., at least one successor state is not visited and its corresponding node is not added to the tree yet; (ii) **Expansion:** in a node that is not fully expanded, an *expansion strategy* is used to choose one or more nodes that will be added to the tree; (iii) **Play-out:** starting from the last node added to the tree a *play-out strategy* chooses which actions to simulate until either a given depth (maximum play-out depth) or a terminal state are reached; and (iv) **Back-propagation:** at the end of the play-out, the result of the simulation is propagated back through all the nodes traversed in the tree and used to update the estimate of their

value. These four phases are repeated until the search budget in terms of time or state evaluations has been exhausted. At this point, the best action in the root node is returned to be played in the real game. Different recommendation policies can be used to decide which action to return and perform, for instance, recommending the one with the highest estimated average score or the one with the highest number of visits.

Many strategies have been proposed for the different phases of MCTS. The standard *selection strategy* is UCT (Upper Confidence bounds applied to Trees) [2]. UCT sees the problem of choosing an action in a certain node of the tree as a Multi-Armed Bandit (MAB) problem and uses the UCB1 [25] sampling strategy to select the action to visit next. UCT selects in node s the action a that maximizes the following formula:

$$UCB1(s, a) = Q(s, a) + K \times \sqrt{\frac{\ln N(s)}{N(s, a)}}, \quad (1)$$

where $Q(s, a)$ is the average result obtained from all the simulations in which action a was played in node (state) s , $N(s)$ is the number of times node s has been visited during the search and $N(s, a)$ is the number of times action a has been selected whenever node s was visited. The K constant is used to control the balance between exploitation of good actions and exploration of less visited ones.

2.3 On-line Parameter Tuning

The parameters of an AI agent can be seen as a vector of integers and doubles (boolean parameters can be handled as integers with only two legal values). The tuning of parameters is therefore a problem of searching optimal numerical vector(s) in a given parameter search space. Given the combinatorial structure of the search space, Sironi and Winands [19] considered the tuning problem as a Combinatorial Multi-Armed Bandit (CMAB) [26]. The definition of the parameter tuning problem as a CMAB is given by the following three components:

- A set of d parameters, $P = \{P_1, \dots, P_d\}$, where each parameter P_i can take m_i different values $V_i = \{v_i^1, \dots, v_i^{m_i}\}$.
- A reward distribution $R : V_1 \times \dots \times V_d \rightarrow \mathbb{R}$ that depends on the combination of values assigned to the parameters.
- A function $L : V_1 \times \dots \times V_d \rightarrow \{true, false\}$ that determines which combinations of parameter values are legal.

In this paper we use the same approach for on-line tuning that was presented in [19]. This approach consists in interleaving MCTS simulations with parameter tuning, as shown in Algorithm 1. Before each MCTS simulation the tuner \mathcal{T} selects a combination of values for the parameters \mathbf{p} . These values are set for the parameters of the agent AI_{MCTS} that performs an MCTS simulation of the game and returns the associated reward. This reward is used by the tuner to update the statistics for the combination of parameters \mathbf{p} . Different allocation strategies can be used by the tuner to decide which parameter combination should be evaluated next depending on these statistics. In this paper we consider the most promising allocation strategy that was introduced in [19], Naïve Monte-Carlo (NMC), and propose two more, one based on an Evolutionary Algorithm and one based on a N-tuple bandit Evolutionary Algorithm.

Algorithm 1 On-line parameter tuning for a given MCTS agent AI_{MCTS} .

Require: G : game to be played

Require: AI_{MCTS} : an agent with parameter vector $\in \mathcal{S}_{AI_{MCTS}}$

Require: \mathcal{T} : tuner

```

1: while time not elapsed do
2:    $\mathbf{p} \leftarrow \mathcal{T}.\text{CHOOSEPARAMVALUES}()$       ▷ Select param. combination using the tuner  $\mathcal{T}$ 
3:    $AI_{MCTS}.\text{set}(\mathbf{p})$                           ▷ Set parameters for  $AI_{MCTS}$ 
4:    $r \leftarrow G.\text{simulate}(AI_{MCTS})$            ▷ Perform MCTS simulation
5:    $\mathcal{T}.\text{UPDATEVALUESSTATS}(\mathbf{p}, r, \dots)$     ▷ Update the statistics, defined by the tuner  $\mathcal{T}$ 

```

3 Allocation Strategies

This section describes the three allocation strategies that are integrated to the GVGAI *sampleMCTS* agent: Naïve Monte-Carlo, Evolutionary Algorithm and N-tuple Bandit Evolutionary Algorithm. As a result, three Self-Adaptive MCTS (SA-MCTS) agents are created. The NMC strategy is the same that is presented in [19]. Among the four strategies proposed in the paper we decide to test this one for GVGAI because it was the one that had more promising results when used to tune parameters for GGP.

3.1 Naïve Monte-Carlo

Naïve Monte-Carlo (NMC) was first proposed by Ontañón [26] to be applied to Real-Time Strategy games. This approach proved suitable to deal with combinatorial search spaces, thus in [19] it was applied to the on-line parameter tuning problem, that is characterized by a combinatorial parameter space. NMC is based on the *naïve assumption*, which is the assumption that the reward associated with a combination of parameter values can be approximated by a linear combination of the rewards associated with each of the single values: $R(\mathbf{p} = \langle p_1, \dots, p_d \rangle) \approx \sum_{i=1}^d R_i(p_i)$.

Algorithm 2 gives the pseudo-code for this strategy. NMC considers one global MAB (MAB_g) and n local MABs ($MAB_i, i = 1 \dots d$), one for each parameter. Each arm of MAB_g corresponds to one of the legal combinations of parameter values that have been evaluated so far, while each arm in MAB_i corresponds to one of the legal values for parameter P_i . This allocation strategy alternates between an *exploration* and an *exploitation* phase. For each MCTS simulation a combination of parameter values is selected by exploration with probability ϵ_0 and by exploitation with probability $(1 - \epsilon_0)$. If exploring, the next combination to be evaluated is selected by choosing each parameter value independently from the corresponding MAB_i (note that a combination that has never been visited before can be generated, thus there is exploration of the search space). If exploiting, the next combination is selected from the global MAB, MAB_g (in this case only previously seen combinations will be selected). MAB_g starts with no arms and a new arm is added whenever a new combination is generated using the local MABs. Whenever a combination of values is evaluated, the reward obtained by the corresponding MCTS simulation is used to update both the statistics associated with the global MAB and the ones associated with each of the local MABs. The SA-MCTS agent built using NMC as a tuner is denoted as SA-MCTS_{NMC}.

Algorithm 2 On-line parameter tuning for a given MCTS agent AI_{MCTS} using Naïve Monte-Carlo.

Require: G : game to be played
Require: AI_{MCTS} : MCTS agent with parameter vector $\in \mathcal{S}_{AI_{MCTS}}$
Require: d : number of parameters to be tuned
Require: \mathcal{S} : parameter search space for AI_{MCTS}
Require: ϵ_0 : probability of performing exploration
Require: π_l : policy to select a parameter value from the local MABs
Require: π_g : policy to select a parameter combination from the global MAB

- 1: $MAB_g \leftarrow$ create a MAB with no arms
- 2: **for** $i \leftarrow 1 : d$ **do**
- 3: $MAB_i \leftarrow$ create a MAB for parameter P_i with one arm for each of its possible values
- 4: **while** time not elapsed **do**
- 5: $\mathbf{p} \leftarrow$ CHOOSEPARAMVALUES($\mathcal{S}, \epsilon_0, \pi_l, \pi_g, MAB_g, MAB_1, \dots, MAB_d$)
- 6: $AI_{MCTS}.set(\mathbf{p})$ ▷ Set parameters for AI_{MCTS}
- 7: $r \leftarrow G.simulate(AI_{MCTS})$ ▷ Perform MCTS simulation
- 8: UPDATEVALUESSTATS($\mathbf{p}, r, MAB_g, MAB_1, \dots, MAB_d$)
- 9: **function** CHOOSEPARAMVALUES($\mathcal{S}, \epsilon_0, \pi_l, \pi_g, MAB_g, MAB_1, \dots, MAB_d$)
- 10: $\mathbf{p} \leftarrow$ create empty combination of values
- 11: **if** $RAND(0, 1) < \epsilon_0$ **then** ▷ Exploration
- 12: **for** $i \leftarrow 1 : d$ **do**
- 13: $\mathbf{p}[i] \leftarrow \pi_l.CHOOSEVALUE(\mathcal{S}, MAB_i)$
- 14: $MAB_g.ADD(\mathbf{p})$
- 15: **else** ▷ Exploitation
- 16: $\mathbf{p} \leftarrow \pi_g.CHOOSECOMBINATION(\mathcal{S}, MAB_g)$
- 17: **return** \mathbf{p}
- 18: **function** UPDATEVALUESSTATS($\mathbf{p}, r, MAB_g, MAB_1, \dots, MAB_d$)
- 19: $MAB_g.UPDATEARMSTATS(\mathbf{p}, r)$
- 20: **for** $i \leftarrow 1 : d$ **do**
- 21: $MAB_i.UPDATEARMSTATS(\mathbf{p}[i], r)$

3.2 Evolutionary Algorithm

Genetic Algorithms (GA) achieve overall good performance in General Video Game Playing [17]. However, in the case of on-line parameter tuning, we aim at using a simple algorithm as tuner, making the best use of the time budget to evaluate more MCTS instances with different parameter settings or more times a good MCTS instance. A combination of parameter values is considered to be an individual, where each single parameter is a gene. A simple Evolutionary Algorithm (EA) with λ individuals has been considered for evolving on-line the parameters for an MCTS agent, where the μ elites in the previous generation are kept and $(\lambda - \mu)$ new individuals are reproduced. Each new individual is reproduced with probability p_c by uniformly random crossover between two elites selected uniformly at random, or by uniformly mutating one bit of a randomly selected elite otherwise. When evaluating a population, each individual (i.e. the corresponding parameter combination) is used to control a different MCTS simulation of the game and the outcome of the simulation is considered as the fitness

of the individual. The SA-MCTS agent built using this EA as a tuner is referred to as SA-MCTS_{EA}.

3.3 N-Tuple Bandit Evolutionary Algorithm

The N-Tuple Bandit Evolutionary Algorithm (NTupleBanditEA) is firstly proposed by Kunanusont *et al.* [27] for automatic game parameter tuning where strong stochastic AI agents were used to evaluate the evolved games with uncertainties. Then, it was applied to the GVGAI framework for evolving game rules and parameter setting in games [28]. It makes use of all the statistics of the previously evaluated solutions and balances the exploration and exploitation between evaluating a new generated solution and re-evaluating an existed solution using UCB1. The detailed algorithm is not given in this paper due to lack of space, more can be found in [27]. We apply the NTupleBanditEA to optimizing the parameters on-line by modeling each parameter to be tuned as a 1-tuple and considering their combinations as n -tuples. For this strategy we need to specify the number of neighbors generated at one iteration, n , and the exploration factor, denoted as K_{NEA} (cf. Algorithm 3 in [27]). The SA-MCTS agent built using this NTupleBanditEA as a tuner is referred to as SA-MCTS_{NEA}.

4 Experimental Settings

In this section we introduce the design of the experiments, including games used as test problem, the baseline AI agent to be tuned and the tested tuners.

4.1 Games

Each of the approaches described in Section 3 is tested on all 5 levels of 20 games (Table 1) of the GVGAI framework. The 20 games are same as the ones used by Gaina *et al.* [17] for studying the parameters of a vanilla RHEA. Gaina *et al.* [17] uniformly randomly selected 20 games from a merged list of games exploited by Nelson [29] and Bontrager *et al.* [30] previously, on which the vanilla MCTS agent performs differently. During every game playing, an agent has 40ms per game tick to decide an action. In all the games, there is no draw. A game terminates if the agent wins or loses the game before 2,000 game ticks or the game is forced to terminate as a loss of the agent after 2,000 game ticks. This is the same setting as in the GVGAI Single-Playing Planning competitions. The only difference is that if the agent exceeds the 40ms limit per game tick it will not be disqualified and can still apply its selected move.

4.2 Tuned Agent and Parameters

We consider the single-player *sampleMCTS* agent in the GVGAI framework as the AI_{MCTS} to be tuned, the performance of which mainly relies on two parameters, the maximum play-out depth D and the UCB1 exploration factor K . The heuristic used by the *sampleMCTS* agent for evaluating a game state is defined as follows:

$$Value(GameState) = \begin{cases} score(GameState) - 10,000,000.0 & \text{if a loss,} \\ score(GameState) + 10,000,000.0 & \text{if a win,} \\ score(GameState) & \text{otherwise.} \end{cases} \quad (2)$$

Table 1: The 20 tested games. The authors of [17] have confirmed that some games have been wrongly listed as deterministic games. *Wait for Breakfast* was listed as deterministic game as it has negligible randomness (detailed in Subsection 5.1).

Deterministic games: Bait, Camel Race, Escape, Hungry Birds, Modality	
Stochastic games	
Negligible randomness	Plaque Attack, Wait for Breakfast
Non-deterministic chasing/fleeing behaviors	Chase, Lemmings, Missile Command, Roguelike
Random NPC(s)	Butterflies, Infection, Roguelike
Very stochastic	Aliens, Chopper, Crossfire, Dig Dug, Intersection, Sea Quest, Survive Zombies

Based on this *sampleMCTS*, the SA-MCTS agents are designed. These agents tune D and K on-line considering 15 possible values for each parameter (i.e. 225 possible combinations of parameters). The same state evaluation heuristic (Equation 2) is used by the self-adaptive agents. The SA-MCTS agents are compared to a baseline agent, the default *sampleMCTS*, with a fixed value for D and K . The parameter settings are summarized as follows:

- *sampleMCTS*: $D = 10$, $K = 1.4$ (default setting in GVGAI);
- **SA-MCTS** agents: $D \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$,
 $K \in \{0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0\}$.

4.3 Tuning Strategies

Three SA-MCTS agents are considered in the experiments, one for each of the presented tuning strategies, NMC, EA and NTupleBanditEA. To distinguish them, in the tables and in subsequent sections they are denoted as SA-MCTS_{NMC}, SA-MCTS_{EA} and SA-MCTS_{NEA}, respectively. The following are the settings for the tuning strategies:

- **SA-MCTS_{NMC}**: $\epsilon_0 = 0.75$ (i.e. select next combination using the local MABs with probability 0.75 and the global MAB with probability 0.25), $\pi_g = \text{UCB1}$ policy with exploration factor $K_g = 0.7$, $\pi_l = \text{UCB1}$ policy with exploration factor $K_l = 0.7$ (note that these two exploration factors are distinct from the UCB1 exploration factor to be tuned and used by the *sampleMCTS* agent).
- **SA-MCTS_{EA}**: population size $\lambda = 50$, elite size $\mu = 25$ (lower values for μ were tested when applying this strategy to GGP and none of them outperformed $\mu = 25$), probability of generating an individual by crossover of two parents, $p_c = 0.5$.
- **SA-MCTS_{NEA}**: number of neighbors generated during evolution $n = 5$ (preliminary tests showed that higher values added no benefit), $K_{NEA} = 0.7$ (exploration constant for the UCB1 formula used to compute the value for a parameter combination [27]).

5 Results and Discussion

The results of the designed SA-MCTS agents and the baseline agent are analyzed and discussed in Section 5.1. Section 5.2 illustrates the performance of some static agents

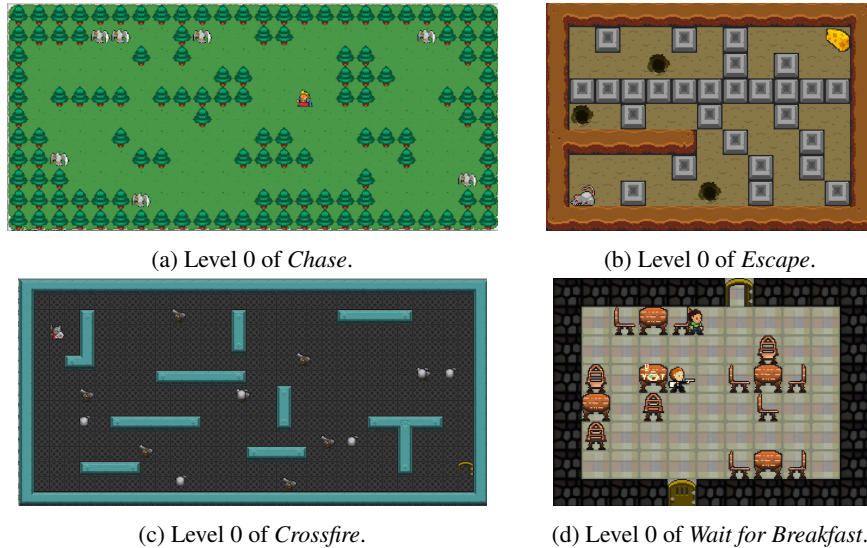


Fig. 1: Screenshots of game screen of *Chase*, *Escape*, *Crossfire* and *Wait for Breakfast*

using constant parameters, the most visited parameter combination during the on-line tuning for each game.

5.1 On-line Tuning Performance

Each of the SA-MCTS agents has been performed on the 5 levels of each of the games 500 times (100 per level), as well as the baseline *sampleMCTS* agent. During every game playing, an agent has 40ms per game tick to decide an action. More about the games and settings has been presented in Subsection 4.1. The win rate and average score for the *sampleMCTS* agent and the SA-MCTS agents on the 20 games are summarized in Tables 2a and 2b, respectively. In our setting, winning a game has the highest priority (Equation 2), thus the win rate is used as the criterion for evaluating a tuner rather than the average score.

The SA-MCTS agents perform overall well on the games where the default *sampleMCTS* also performs well, except for the games of *Chopper* and *Survive Zombies*. Moreover, in some of the games that the *sampleMCTS* has poor performance, e.g. *Chase*, *Escape*, *Crossfire* and *Wait for Breakfast* (Fig.1), the SA-MCTS agents significantly improve the win rate. These four games are detailed below.

Chase. (Fig. 1(a)) The player must chase and kill scared goats that flee from the player. A dead goat turns to a corpse immediately and the player gains 1 point as score. A goat becomes angry as soon as it finds another goat's corpse, and starts to chase the player. The player wins the game if all scared goats are dead, but it will lose one point and loses the game immediately if is caught by an angry goat. The game is very difficult as an angry goat will never turn back to a normal one, and by default the game ends with a loss of the player after running 2, 000 game ticks. Thus, once a goat becomes angry,

Table 2: Average win rate (%) and average game score over 5 levels of each game for the *sampleMCTS* agent and the SA-MCTS agents. The best values are in bold.

(a) Average win rate (%) over 5 levels of each game.

Games	sampleMCTS	SA-MCTS _{NMC}	SA-MCTS _{EA}	SA-MCTS _{NEA}
Aliens	100.0(±0.00)	99.8(±0.39)	100.0(±0.00)	99.4(±0.68)
Bait	6.6(±2.18)	7.0(±2.24)	7.8(±2.35)	8.4(±2.43)
Butterflies	95.2(±1.88)	95.0(±1.91)	94.2(±2.05)	95.4(±1.84)
Camel Race	4.2(±1.76)	4.6(±1.84)	6.2(±2.12)	5.2(±1.95)
Chase	3.2(±1.54)	7.2(±2.27)	9.2(±2.54)	7.4(±2.30)
Chopper	91.4(±2.46)	88.6(±2.79)	83.2(±3.28)	50.8(±4.39)
Crossfire	4.2(±1.76)	11.6(±2.81)	11.4(±2.79)	15.6(±3.18)
Dig Dug	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)
Escape	0.2(±0.39)	4.4(±1.80)	7.6(±2.33)	13.0(±2.95)
Hungry Birds	5.4(±1.98)	2.6(±1.40)	4.6(±1.84)	3.8(±1.68)
Infection	97.0(±1.50)	95.6(±1.80)	97.6(±1.34)	97.8(±1.29)
Intersection	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)
Lemmings	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)
Missile Command	60.4(±4.29)	60.8(±4.28)	58.0(±4.33)	58.6(±4.32)
Modality	27.0(±3.90)	27.4(±3.91)	26.0(±3.85)	28.4(±3.96)
Plaque Attack	91.8(±2.41)	92.0(±2.38)	92.8(±2.27)	92.6(±2.30)
Roguelike	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)
Sea Quest	55.0(±4.37)	47.8(±4.38)	55.6(±4.36)	43.2(±4.35)
Survive Zombies	41.0(±4.32)	41.0(±4.32)	34.8(±4.18)	34.8(±4.18)
Wait for Breakfast	15.4(±3.17)	20.4(±3.54)	28.8(±3.97)	44.0(±4.36)
Avg Win%	39.9(±0.96)	40.3(±0.96)	40.9(±0.96)	39.9(±0.96)

(b) Average game score over 5 levels of each game. Note that in GVGAI, winning a game with low score is higher ranked than losing a game with a high score.

Games	sampleMCTS	SA-MCTS _{NMC}	SA-MCTS _{EA}	SA-MCTS _{NEA}
Aliens	67.8(±1.27)	64.5(±1.17)	64.4(±1.18)	65.6(±1.23)
Bait	2.6(±0.27)	4.5(±0.55)	2.3(±0.28)	5.6(±0.63)
Butterflies	30.3(±1.34)	30.4(±1.32)	30.0(±1.27)	31.1(±1.33)
Camel Race	-0.8(±0.05)	-0.8(±0.05)	-0.7(±0.05)	-0.7(±0.05)
Chase	2.7(±0.18)	3.1(±0.18)	3.1(±0.19)	3.2(±0.20)
Chopper	11.4(±0.55)	10.8(±0.56)	9.7(±0.65)	4.4(±0.73)
Crossfire	0.1(±0.09)	0.3(±0.15)	0.2(±0.16)	0.4(±0.18)
Dig Dug	11.2(±0.79)	10.1(±0.83)	10.3(±0.81)	9.0(±0.74)
Escape	0.0(±0.00)	0.0(±0.02)	0.1(±0.02)	0.1(±0.03)
Hungry Birds	7.4(±2.09)	3.5(±1.48)	5.3(±1.88)	4.4(±1.72)
Infection	14.3(±0.71)	13.4(±0.69)	14.5(±0.75)	14.1(±0.74)
Intersection	1.0(±0.0)	1.1(±0.09)	2.1(±0.27)	1.0(±0.04)
Lemmings	-3.5(±0.30)	-2.3(±0.23)	-4.3(±0.34)	-1.4(±0.16)
Missile Command	4.4(±0.44)	4.2(±0.45)	4.0(±0.46)	3.9(±0.44)
Modality	0.3(±0.04)	0.3(±0.04)	0.3(±0.04)	0.3(±0.04)
Plaque Attack	46.9(±1.64)	46.4(±1.54)	50.4(±1.68)	48.6(±1.60)
Roguelike	3.5(±0.41)	2.9(±0.38)	3.2(±0.40)	3.2(±0.39)
Sea Quest	1734.6(±169.97)	1575.1(±163.41)	1774.2(±167.22)	1288.1(±135.44)
Survive Zombies	2.6(±0.30)	2.7(±0.31)	2.2(±0.29)	2.2(±0.29)
Wait for Breakfast	0.2(±0.03)	0.2(±0.04)	0.3(±0.04)	0.4(±0.04)
Avg Score	96.8(±11.25)	88.5(±10.56)	98.6(±11.26)	74.2(±8.70)

it will inevitably lead to a lost game for the player, but this negative reward is delayed until the end of the game. In our context, the game rules are not given to the agent, so the agent will not be aware of the defeat until being caught or after 2,000 game ticks. The baseline agent, *sampleMCTS* only wins 32 games out of 500, while the SA-MCTS agents win at least 72 games.

Escape. (Fig. 1(b)) It is a puzzle game with wide search space and required long-term planning. The player (rat) wins the game by taking the cheese. The player’s score is 1 if it wins, -1 otherwise. Sometimes, the player needs to push a block *into* a hole in order to *clear* a path to the cheese. Each of the 3 on-line tuning agents greatly improves the win rate (at least 22 times higher) compared to the baseline agent, in particular, SA-MCTS_{NEA}, increases the win rate from 0.2% to 13.0%. In a similar tested puzzle game *Bait*, the player needs to push a block to *fill* a hole in order to *build* a path to the cheese. Interestingly, the win rate on *Bait* is not that highly improved, though some significant improvements have been observed.

Crossfire. (Fig. 1(c)) The player wins the game if it reaches the exit door without being hit by any shell and gets 5 as game score. Once the player is hit by a shell, the game ends immediately with a loss of the player and -1 as game score. The win rates achieved by the SA-MCTS agents are at least 2 times higher than the one by the baseline agent.

Wait for Breakfast. (Fig. 1(d)) In this game, all tables are empty when the game starts, a waiter (NPC in black in Fig. 1(d)) serves a breakfast to the table with only one chair at a random time. The player (avatar in green in Fig. 1(d)) wins the game only if it sits on the chair on the table after the breakfast is served, otherwise (taking a wrong chair or taking the right chair before the breakfast is served), it loses the game. When the waiter serves the breakfast is defined as: at any game tick, if no breakfast is served yet, the waiter serves a breakfast with probability 0.05; the waiter serves at most one breakfast during a whole game playing. The probability of no breakfast has been served 10 game ticks after starting a game is $0.05^{10} = 9.7656e-14$. This game can be considered as a deterministic game. The win rate is significantly improved by all the 3 SA-MCTS agents, among which SA-MCTS_{NEA} increases the win rate from 15.4% to 44.0%.

For reference, the average of median numbers of iterations per game tick for all tested agents are given in Table 3. Note that during one iteration of any of the agents, the forward model is called multiple times.

The most visited combination of the UCB1 exploration factor K and the maximum play-out depth D per game (over 500 runs) for each of the SA-MCTS agents are extracted and listed in Table 4. Surprisingly, the most used play-out depth is 1. The SA-MCTS agents prefers the averaged instant reward than the averaged long-term reward. A possible reason is the heuristic (Equation 2) used by the agents. Assuming an MCTS agent with maximum play-out depth 10, even for a deterministic game, the number of possible game states after a play-out can increase at most exponentially, the reward after a play-out can vary between a large range due to the same reason. If it is a loss after a play-out, then the average reward obtained by the parameter combinations with $D = 10$ will decrease a lot due to the 10,000,000.0 penalty in score; if it is a win, then the average reward will increase thanks to the 10,000,000.0 award in score. In the games where a SA-MCTS agent gets a very low win rate, the parameter combinations with $D = 10$ are more likely to have an overall low average reward and prefer a lower

Table 3: Average median number of iterations per tick for the *sampleMCTS* agent and the SA-MCTS agents. The number of forward model calls per iteration depends on the tuner and is sometimes not a constant. For space reasons, headers have been shortened as follows: SA-MCTS_{NMC} = NMC, SA-MCTS_{EA} = EA, SA-MCTS_{NEA} = NEA.

Games	sampleMCTS	NMC	EA	NEA	Games	sampleMCTS	NMC	EA	NEA
Aliens	35.16	41.80	25.46	50.46	Infection	23.77	25.50	21.24	23.72
Bait	70.83	123.71	86.88	154.77	Intersection	35.66	46.83	68.96	39.93
Butterflies	26.97	29.54	24.01	29.32	Lemmings	22.24	36.57	69.07	62.04
Camel Race	21.88	24.55	24.55	24.55	Missile Command	39.08	43.12	52.30	43.78
Chase	29.33	37.49	36.98	45.46	Modality	96.49	104.80	108.51	103.09
Chopper	17.30	22.60	23.63	50.71	Plaque Attack	15.92	18.43	14.75	17.70
Crossfire	19.55	33.68	45.74	52.46	Roguelike	15.45	19.54	20.96	27.61
Dig Dug	14.17	20.24	25.25	34.21	Sea Quest	34.25	45.47	28.55	79.28
Escape	37.75	69.86	96.02	119.70	Survive Zombies	18.43	30.80	47.85	53.12
Hungry Birds	46.37	50.56	52.34	51.32	Wait for Breakfast	83.96	106.30	158.73	178.17

Table 4: Most visited combination of parameters per game for each of the SA-MCTS agents. Parameter combination are expressed with the format $[K, D]$. K refers to the UCB1 exploration factor and D is the maximum play-out depth.

Games	SA-MCTS _{NMC}	SA-MCTS _{EA}	SA-MCTS _{NEA}	Games	SA-MCTS _{NMC}	SA-MCTS _{EA}	SA-MCTS _{NEA}
Aliens	[1.4, 1.0]	[0.6, 15.0]	[0.6, 1.0]	Infection	[1.8, 2.0]	[0.6, 15.0]	[2.0, 15.0]
Bait	[1.1, 1.0]	[0.7, 1.0]	[0.7, 1.0]	Intersection	[1.3, 1.0]	[0.7, 1.0]	[1.2, 15.0]
Butterflies	[1.3, 3.0]	[0.6, 15.0]	[1.6, 13.0]	Lemmings	[0.6, 1.0]	[0.6, 1.0]	[0.6, 1.0]
Camel Race	[1.8, 1.0]	[2.0, 4.0]	[1.4, 14.0]	Missile Command	[0.6, 12.0]	[0.7, 1.0]	[0.8, 15.0]
Chase	[0.7, 1.0]	[0.6, 1.0]	[0.7, 1.0]	Modality	[1.1, 1.0]	[1.0, 4.0]	[0.8, 13.0]
Chopper	[0.6, 1.0]	[0.7, 1.0]	[0.8, 1.0]	Plaque Attack	[0.8, 4.0]	[0.7, 15.0]	[0.7, 15.0]
Crossfire	[0.7, 1.0]	[0.6, 1.0]	[0.7, 1.0]	Roguelike	[0.8, 1.0]	[0.6, 1.0]	[0.7, 1.0]
Dig Dug	[0.7, 1.0]	[0.7, 1.0]	[0.6, 1.0]	Sea Quest	[0.6, 1.0]	[0.6, 15.0]	[1.0, 1.0]
Escape	[1.1, 1.0]	[0.6, 1.0]	[0.6, 1.0]	Survive Zombies	[0.7, 1.0]	[0.6, 1.0]	[0.7, 1.0]
Hungry Birds	[1.4, 13.0]	[2.0, 1.0]	[1.0, 1.0]	Wait for Breakfast	[1.0, 1.0]	[1.0, 1.0]	[0.7, 1.0]

maximum play-out depth D , whereas in the games where a SA-MCTS agent gets a high win rate, the parameter combinations with higher D are favorable. For instance, in the game *Plaque Attack*, all agents achieve a win rate higher than 90%, the most visited maximum play-out depth D is 4, 15, and 15 for SA-MCTS_{NMC}, SA-MCTS_{EA} and SA-MCTS_{NEA} respectively.

5.2 On-line Tuning Validation

The most visited parameter combinations during learning are set to the *sampleMCTS* and tested on the same set of games. The parameters are fixed during game playing, so no more tuning happened. We denote these instances of *sampleMCTS* as *instance_{NMC}*, *instance_{EA}* and *instance_{NEA}*. The average win rate and average final score for the *sampleMCTS* agent with default parameter values and the *sampleMCTS* instances, *sampleMCTS* agents with the most visited combination per game by each of the SA-MCTS agents (cf. Table 4) are presented in Tables 5a and 5b, respectively.

In the game *Escape* (Fig. 1(b)), the win rate is significantly improved from 0.2% (baseline agent) to 30.8% by the *instance_{NEA}* with parameters tuned by *NEA*, while the highest win rate of on-line tuning agents is 13.0% by *sampleMCTS_{NEA}* (shown in Table 2). In the game *Wait for Breakfast* (Fig. 1(d)), the win rate is significantly im-

Table 5: Average win rate (%) and average game score over 5 levels for the *sampleMCTS* agent with default parameter values and the *sampleMCTS* agents with the most visited combination per game by each of the SA-MCTS agents. The best values are in bold.

(a) Average win rate (%) over 5 levels for each game

Games	sampleMCTS	instance _{NMC}	instance _{EA}	instance _{NEA}
Aliens	100.0(±0.00)	68.0(±4.09)	100.0(±0.00)	69.6(±4.04)
Bait	6.6(±2.18)	4.2(±1.76)	3.6(±1.63)	3.6(±1.63)
Butterflies	95.2(±1.88)	93.0(±2.24)	95.4(±1.84)	95.0(±1.91)
Camel Race	4.2(±1.76)	3.8(±1.68)	6.0(±2.08)	4.2(±1.76)
Chase	3.2(±1.54)	6.2(±2.12)	7.4(±2.30)	6.2(±2.12)
Chopper	91.4(±2.46)	0.0(±0.00)	0.0(±0.00)	0.2(±0.39)
Crossfire	4.2(±1.76)	9.8(±2.61)	9.2(±2.54)	9.8(±2.61)
Dig Dug	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)
Escape	0.2(±0.39)	29.4(±4.00)	30.8(±4.05)	30.8(±4.05)
Hungry Birds	5.4(±1.98)	5.0(±1.91)	1.4(±1.03)	2.8(±1.45)
Infection	97.0(±1.50)	97.2(±1.45)	97.8(±1.29)	96.6(±1.59)
Intersection	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)	100.0(±0.00)
Lemmings	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)
Missile Command	60.4(±4.29)	64.2(±4.21)	31.8(±4.09)	64.2(±4.21)
Modality	27.0(±3.90)	16.0(±3.22)	25.4(±3.82)	27.2(±3.90)
Plaque Attack	91.8(±2.41)	67.2(±4.12)	96.0(±1.72)	96.0(±1.72)
Roguelike	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)	0.0(±0.00)
Sea Quest	55.0(±4.37)	18.2(±3.39)	58.4(±4.32)	18.2(±3.39)
Survive Zombies	41.0(±4.32)	28.8(±3.97)	25.4(±3.82)	28.8(±3.97)
Wait for Breakfast	15.4(±3.17)	60.8(±4.28)	60.8(±4.28)	60.2(±4.29)
Avg Win%	39.9(±0.96)	33.6(±0.93)	37.5(±0.95)	35.7(±0.94)

(b) Average score over 5 levels for each game. Note that in GVGAI, winning a game with low score is higher ranked than losing a game with a high score.

Games	sampleMCTS	instance _{NMC}	instance _{EA}	instance _{NEA}
Aliens	67.8(±1.27)	55.9(±0.90)	64.1(±1.16)	55.8(±0.96)
Bait	2.6(±0.27)	3.0(±0.40)	3.1(±0.41)	3.1(±0.41)
Butterflies	30.3(±1.34)	31.1(±1.30)	30.7(±1.29)	30.8(±1.31)
Camel Race	-0.8(±0.05)	-0.8(±0.04)	-0.7(±0.05)	-0.8(±0.05)
Chase	2.7(±0.18)	2.7(±0.20)	3.0(±0.20)	2.7(±0.20)
Chopper	11.4(±0.55)	-10.6(±0.33)	-10.8(±0.34)	-10.8(±0.33)
Crossfire	0.1(±0.09)	-0.3(±0.15)	-0.3(±0.15)	-0.3(±0.15)
Dig Dug	11.2(±0.79)	4.9(±0.53)	4.9(±0.53)	4.9(±0.52)
Escape	0.0(±0.00)	0.3(±0.04)	0.3(±0.04)	0.3(±0.04)
Hungry Birds	7.4(±2.09)	6.7(±2.01)	2.0(±1.10)	3.0(±1.47)
Infection	14.3(±0.71)	12.7(±0.65)	14.0(±0.70)	12.9(±0.62)
Intersection	1.0(±0.00)	6.4(±0.64)	6.6(±0.63)	1.0(±0.00)
Lemmings	-3.5(±0.30)	-0.1(±0.03)	-0.1(±0.03)	-0.1(±0.03)
Missile Command	4.4(±0.44)	4.5(±0.44)	0.7(±0.33)	4.6(±0.46)
Modality	0.3(±0.04)	0.2(±0.03)	0.3(±0.04)	0.3(±0.04)
Plaque Attack	46.9(±1.64)	31.4(±1.36)	52.6(±1.59)	52.6(±1.59)
Roguelike	3.5(±0.41)	1.8(±0.28)	1.6(±0.26)	1.7(±0.29)
Sea Quest	1734.6(±169.97)	591.2(±98.53)	1891.8(±177.09)	583.7(±95.36)
Survive Zombies	2.6(±0.30)	2.0(±0.29)	1.8(±0.28)	2.0(±0.29)
Wait for Breakfast	0.2(±0.03)	0.6(±0.04)	0.6(±0.04)	0.6(±0.04)
Avg Score	96.8(±11.25)	37.2(±5.53)	103.3(±11.96)	37.4(±5.37)

proved from 15.4% (baseline) to 60.8% by the *instance_{NMC}* and *instance_{EA}*, while the highest win rate obtained by the on-line tuning agents is 44.0% by *sampleMCTS_{NEA}* (shown in Table 2). However, some instances with constant parameter values performed much worse than the baseline agent in some games. For instance, in the game *Aliens*, the baseline agent ($D = 10$) and *instance_{EA}* ($D = 15$) win all the games, while *instance_{NMC}* and *instance_{NEA}* win $\sim 68\%$ games due to the maximum play-out depth $D = 1$. The same scenarios happen in the games *Sea Quest* and *Survive Zombies*. Due to the maximum play-out depth $D = 1$, *instance_{NMC}*, *instance_{EA}* and *instance_{NEA}* lose more often the puzzle game *Bait* and lose almost all the 500 runs of *Chopper*. Our SA-MCTS agents are more robust than the non-SA MCTS instances with constant parameter values.

6 Conclusion and Future Work

General Video Game Playing has attracted interest from researchers during the last years. On-line tuning an agent for GVGP provides more adaptive and robust agents, however, it is rather difficult due to the real-time setting. In this paper, we have incorporated three different algorithms, Naïve Monte-Carlo, the Evolutionary Algorithm, and N-Tuple Bandit Evolutionary Algorithm, to tune on-line the *sampleMCTS* agent in the GVGAI framework and create three Self-Adaptive MCTS (SA-MCTS) agents. The SA-MCTS agents have been compared to the baseline MCTS agent, *sampleMCTS*, on 20 single-player GVGAI games. The SA-MCTS agents perform similarly to the *sampleMCTS* on the games that *sampleMCTS* performs well, and significantly improve the win rate in the games that *sampleMCTS* perform poorly. The SA-MCTS agents achieve more robust results on the tested games. Additionally, the *sampleMCTS* instances using most sampled parameter settings by each of the three SA-MCTS agents per game improve the win rate on *Wait for Breakfast* and *Escape* by 4 times and 150 times, respectively.

The approaches used in this paper and the tested tuning strategies have been a success in GVGAI, in particular the tuning strategy NMC has also obtained promising results in GGP. On-line agent tuning for GVGP is important because successful approaches can rapidly improve and specialize the general abilities of the agents, leading to better performance across a wide range of games. The research has application outside of games to any problem that has a fast simulation model and requires rapid and adaptive decision making.

This work can be extended in different directions. Applying the SA-MCTS agents for playing 2-player GVGAI games, where the time limit remains $40ms$, will be interesting. The heuristic (Equation 2) of *sampleMCTS* is directly used as the reward for tuners. The preference of maximum play-out depth 1 encourages us to explore a better reward function for the tuners. In this paper, we focus on the discrete search space, the search space of the UCB1 exploration factor is discretized by being uniformly sampled within a range using a fixed gap, though that was just an experimental design choice: all algorithms under test can work with any selection of parameter choices. An interesting future work is applying continuous parameter tuning using Evolutionary Strategies (ES), such as the Covariance Matrix Adaptation Evolution Strategy (CMA-ES), which

does not rely on the assumption of smooth problem. Another work in the future is tuning a more advanced agent possibly with more parameters to be tuned. A potentially good choice is the winner agent of the 2016 GVGAI Single-Player Planning Championship, *MaastCTS2* [16], which is also MCTS-based. There is also much work to be done in tuning the play-out policy. A particular challenge is to deal more efficiently with flat reward landscapes using methods that seek diverse points in the (game) state space in the absence of any differences in the explicit reward (e.g. Novelty Search). Tuning a non-MCTS-based agent, such as an Rolling Horizon Evolutionary Algorithm, will be interesting and challenging due to the real-time control in evaluating a population. In general, agents with more parameters to be optimized will provide a more challenging test for the sample efficiency of the tuning methods.

Acknowledgments. This work is partially funded by the Netherlands Organisation for Scientific Research (NWO) in the framework of the project GoGeneral, grant number 612.001.121, and the EPSRC IGGI Centre for Doctoral Training, grant number EP/L015846/1.

References

1. Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo tree search. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M. (eds.) Proceedings of the 5th International Conference on Computer and Games. Lecture Notes in Computer Science (LNCS), vol. 4630, pp. 72–83. Springer-Verlag, Heidelberg, Germany (2007)
2. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: Machine Learning: ECML 2006, LNCS, vol. 4212, pp. 282–293. Springer (2006)
3. Yannakakis, G.N., Togelius, J.: Artificial Intelligence and Games. Springer (2018), <http://gameaibook.org>
4. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of Monte Carlo tree search methods. Computational Intelligence and AI in Games, IEEE Transactions on 4(1), 1–43 (2012)
5. Helmbold, D.P., Parker-Wood, A.: All-moves-as-first heuristics in Monte-Carlo Go. In: IC-AI. pp. 605–610 (2009)
6. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: Proceedings of the 24th International Conference on Machine Learning. pp. 273–280. ACM (2007)
7. Cazenave, T.: Generalized rapid action value estimation. In: Proceedings of the 24th International Joint Conference on Artificial Intelligence. pp. 754–760. AAAI Press (2015)
8. Sironi, C.F., Winands, M.H.M.: Comparison of rapid action value estimation variants for general game playing. In: Computational Intelligence and Games (CIG), 2016 IEEE Conference on. pp. 309–316. IEEE (2016)
9. Finnsson, H., Björnsson, Y.: Simulation-based approach to general game playing. In: AAAI. vol. 8, pp. 259–264 (2008)
10. Powley, E.J., Cowling, P.I., Whitehouse, D.: Information capture and reuse strategies in Monte Carlo tree search, with applications to games of hidden information. Artificial Intelligence 217, 92–116 (2014)
11. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al.: Mastering the game of Go with deep neural networks and tree search. Nature 529(7587), 484–489 (2016)

12. Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al.: Mastering the game of Go without human knowledge. *Nature* 550(7676), 354–359 (2017)
13. Björnsson, Y., Finnsson, H.: CadiaPlayer: a simulation-based general game player. *Computational Intelligence and AI in Games*, IEEE Transactions on 1(1), 4–15 (2009)
14. Perez-Liebana, D., Samothrakis, S., Togelius, J., Schaul, T., Lucas, S.M., Couëtoux, A., Lee, J., Lim, C.U., Thompson, T.: The 2014 general video game playing competition. *IEEE Transactions on Computational Intelligence and AI in Games* 8(3), 229–243 (2016)
15. Gaina, R.D., Couëtoux, A., Soemers, D.J.N.J., Winands, M.H.M., Vodopivec, T., Kirchgeßner, F., Liu, J., Lucas, S.M., Perez-Liebana, D.: The 2016 two-player GVGAI competition. *IEEE Transactions on Computational Intelligence and AI in Games* (2017), accepted for publication.
16. Soemers, D.J.N.J., Sironi, C.F., Schuster, T., Winands, M.H.M.: Enhancements for real-time Monte-Carlo tree search in general video game playing. In: *Computational Intelligence and Games (CIG)*, 2016 IEEE Conference on. pp. 1–8. IEEE (2016)
17. Gaina, R.D., Liu, J., Lucas, S.M., Pérez-Liebana, D.: Analysis of vanilla rolling horizon evolution parameters in general video game playing. In: *European Conference on the Applications of Evolutionary Computation*. pp. 418–434. Springer (2017)
18. Bravi, I., Khalifa, A., Holmgård, C., Togelius, J.: Evolving game-specific UCB alternatives for general video game playing. In: *European Conference on the Applications of Evolutionary Computation*. pp. 393–406. Springer (2017)
19. Sironi, C.F., Winands, M.H.M.: On-line parameters tuning for Monte-Carlo tree search in general game playing. In: *6th Workshop on Computer Games (CGW)* (2017)
20. Liu, J., Perez-Liebana, D., Lucas, S.M.: The single-player GVGAI learning framework - technical manual (2017)
21. Khalifa, A., Perez-Liebana, D., Lucas, S.M., Togelius, J.: General video game level generation. In: *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*. pp. 253–259. ACM (2016)
22. Khalifa, A., Green, M.C., Perez-Liebana, D., Togelius, J.: General video game rule generation. In: *Computational Intelligence and Games (CIG)*, 2017 IEEE Conference on. pp. 170–177. IEEE (2017)
23. Ebner, M., Levine, J., Lucas, S.M., Schaul, T., Thompson, T., Togelius, J.: Towards a video game description language. In: *Dagstuhl Follow-Ups*. vol. 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2013)
24. Bellemare, M.G., Naddaf, Y., Veness, J., Bowling, M.: The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.(JAIR)* 47, 253–279 (2013)
25. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. *Machine learning* 47(2-3), 235–256 (2002)
26. Ontanón, S.: Combinatorial multi-armed bandits for real-time strategy games. *Journal of Artificial Intelligence Research* 58, 665–702 (2017)
27. Kunanusont, K., Gaina, R.D., Liu, J., Perez-Liebana, D., Lucas, S.M.: The n-tuple bandit evolutionary algorithm for automatic game improvement. In: *Evolutionary Computation (CEC)*, 2017 IEEE Congress on. IEEE (2017)
28. Perez-Liebana, D., Liu, J., Lucas, S.M.: General video game AI as a tool for game design. Tutorial at *IEEE Conference on Computational Intelligence and Games (CIG)* (2017)
29. Nelson, M.J.: Investigating vanilla MCTS scaling on the GVG-AI game corpus. In: *Proceedings of the 2016 IEEE Conference on Computational Intelligence and Games*. pp. 403–409 (2016)
30. Bontrager, P., Khalifa, A., Mendes, A., Togelius, J.: Matching games and algorithms for general video game playing. In: *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*. pp. 122–128 (2016)