This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TG.2018.2884768, IEEE Transactions on Games

Self-Adaptive Monte-Carlo Tree Search in General Game Playing

Chiara F. Sironi, Jialin Liu, *Member, IEEE,* Mark H. M. Winands, *Member, IEEE*

***Abstract*—Many enhancements for Monte-Carlo Tree Search (MCTS) have been applied successfully in General Game Playing (GGP). MCTS and its enhancements are controlled by multiple parameters that require extensive and time-consuming off-line optimization. Moreover, as the played games are unknown in advance, off-line optimization cannot tune parameters specifically for single games. This article proposes a self-adaptive MCTS strategy (SA-MCTS) that integrates within the search a method to automatically tune search-control parameters on-line per game. It presents five different allocation strategies that decide how to allocate available samples to evaluate parameter values. Experiments with 1s play-clock on multi-player games show that for all the allocation strategies the performance of SA-MCTS that tunes two parameters is at least equal to or better than the performance of MCTS tuned off-line and not optimized per-game. The allocation strategy that performs the best is NTBEA. This strategy also achieves a good performance when tuning four parameters. SA-MCTS can be considered as a successful strategy for domains that require parameter tuning for every single problem, and it is also a valid alternative for domains where off-line parameter tuning is costly or infeasible.**

I. Introduction

MONTE-CARLO Tree Search (MCTS) [1], [2] is a simulation-based search technique. It has become popular in game playing, having particular success in General Game Playing (GGP) [3]. GGP aims at creating agents that play any abstract game by only being given its rules, without using prior knowledge, and usually having only a few seconds available to select which moves to play.

Search-control strategies and enhancements have been proposed for MCTS in various domains [4]. Among the most successful ones are Rapid Action Value Estimation (RAVE) [3], [5], its generalization, GRAVE [6], and the Move Average Sampling Technique (MAST) [3]. These strategies improve different parts of the search by exploiting information about general performance of the moves.

The behavior of MCTS strategies is normally controlled by a certain number of parameters. The performance of these strategies depends on how parameter values are set. Sometimes, extensive off-line tuning is required to find the best values. Parameters might also be inter-constrained, so either a large amount of time is spent testing all possible combinations of values or the parameters are tuned separately ignoring the inter-dependency. Research has also shown that in some contexts the best values could be game dependent [6], [7] and tuning parameters per game might improve the performance.

In the context of GGP, off-line tuning of parameters per game is infeasible because agents have to deal with a theoretically unlimited number of games, treating each of them as a new game that they have never seen before. This is why off-line parameter tuning in GGP usually looks for a single combination of values to use for all games, picking the one that performs overall best on a certain (preferably heterogeneous) set of benchmark games. Tuning search-control parameters for each game in GGP is still possible if done on-line. A Self-adaptive MCTS (SA-MCTS) can be designed by devising an on-line tuning method that adjusts the parameter values for each new game being played. Such method should also aim at tuning the parameters in combination, because parameter values are usually interdependent. In this case, the number of possible combinations of parameters can become very large. Therefore, an efficient strategy has to be designed to decide how to allocate the available samples to test them.

This article extends on the authors' previous work [8]. It describes the same on-line tuning method used to achieve a SA-MCTS strategy and formulates the tuning problem as a Combinatorial Multi-Armed Bandit (CMAB) [9]. Five allocation strategies are also presented: Naïve Monte-Carlo (NMC) [9], [10], Linear Side Information (LSI) [11], an Evolutionary Algorithm (EA), the recently proposed N-Tuple Bandit Evolutionary Algorithm (NTBEA) [12], [13] and the popular Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) [14]. NMC and LSI have been already presented in [8] and in this article are tested with some improvements and better tuned parameters. EA, NTBEA and CMA-ES are applied to on-line parameter tuning in GGP for the first time. With CMA-ES this article tests an allocation strategy that considers a continuous parameter domain, while in [8] only strategies with a discrete parameter domain were considered. Moreover, this article evaluates the robustness of the allocation strategies by testing them on MCTS-based agents with different skills, on increasing number of tuned parameters and, when tuning six parameters, for different time constraints.

C. F. Sironi and M. H. M. Winands are with the Games and AI Group, Department of Data Science and Knowledge Engineering, Maastricht University, P.O. Box 616, 6200 MD, Maastricht, The Netherlands (email: {c.sironi,m.winands}@maastrichtuniversity.nl).

J. Liu is with the Shenzhen Key Laboratory of Computational Intelligence, University Key Laboratory of Evolving Intelligent Systems of Guangdong Province, Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China (email: liujl@sustc.edu.cn).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TG.2018.2884768, IEEE Transactions on Games

The remainder of this article is structured as follows. Section II introduces previous work related to parameter tuning. Section III gives background on the Multi-Armed Bandit (MAB) problem, evolutionary computation techniques and MCTS. How to obtain a SA-MCTS algorithm using on-line parameter tuning and how to formulate the tuning problem is discussed in Section IV. Section V describes the five allocation strategies. Results obtained by testing these strategies in the context of GGP are presented in Section VI, and Section VII gives the conclusion and outlines possible future work.

II. Related Work

In the area of game playing on-line tuning of search parameters has not been well explored. Recently, for General Video Game Playing (GVGP) [15] SA-MCTS agents with on-line parameter tuning have been proposed [16]. These agents achieve robust results, increasing the performance for a few of the tested games. However, the nature of GVGP games and the search time of $40ms$, used to ensure real-time play, are different from GGP, which focuses on board games and allocates a few seconds per move.

More attention has been given to automatic off-line tuning of search parameters (e.g. [17]–[19]). What most of these methods have in common is the need for a high number of samples (e.g. game simulations) against a benchmark player in order to find an optimal parameter configuration, which is then evaluated by playing against an "identical" agent with manually tuned parameters.

Other research focuses on designing Hyper-Heuristics, which are *"a search method or learning mechanism for selecting or generating heuristics to solve computational search problems"* [20]. This concept has been applied to devise an hyper-agent [21] for the GVGP framework [22]. The agent's hyper-heuristic is trained off-line to recognize from a portfolio of sub-agents the best one for the game at hand. The hyper-heuristic approach presented in [23], instead, devises an on-line mechanism to select from a portfolio of strategies the one that is best suited for the current game. A similar concept is *algorithm selection* [24], which consists in choosing the most appropriate algorithm for the instance at hand, given a set of problem instances and a set of algorithms that present a varying performance depending on the instance they are applied to.

The work proposed in this article is similar to the idea of hyper-heuristic and algorithm selection. Some parameters can decide whether to (de)activate a certain search-control strategy depending on the value that is assigned to them, actually originating multiple search algorithms. Parameter tuning can be seen as a hyper-heuristic to choose which strategies or algorithms to use from a portfolio determined by the available parameter configurations.

III. Background

This section provides background on the MAB problem, on evolutionary computation techniques and on MCTS.

A. Multi-Armed Bandit

The MAB problem [25] with n arms is defined as a set of n unknown independent real reward distributions $R = \{R_1, ..., R_n\}$, each associated to one of the arms. When an arm is played a reward is obtained as a sample of the corresponding distribution. The aim of a sampling strategy for a MAB is to maximize the cumulative reward obtained by successive plays of the arms. For each iteration the strategy chooses which arm to play depending on past played arms and obtained rewards.

B. Evolutionary Computation Techniques

Evolutionary computation [26] is a set of nature-inspired optimization algorithms. In general, a population of individuals (candidate solutions) is randomly initialized when the optimization process starts, then updated iteratively by evaluation, selection and reproduction until stopping conditions are met. The population can be composed by only one individual. Evolutionary computation techniques are classified by different ways of reproduction, size of population, discrete or continuous search space, etc. Most of the applications of evolutionary computation techniques to game playing focus on evolving sequences of actions to play and evolving game parameters or levels [12], [27].

C. Monte-Carlo Tree Search

MCTS is a best-first search algorithm that incrementally builds a tree representation of the search space of a game and uses simulations to estimate the values of game states [1], [2]. Four phases can be identified for each iteration of the MCTS algorithm:

Selection: a *selection strategy* is used at every node in the tree to select the next move to visit until a node is reached that is not fully expanded (i.e. not for all the successors states a node has been added to the tree).

Expansion: one or more nodes are added to the tree according to a given *expansion strategy*.

Play-out: starting from the last node added to the tree a *play-out* strategy chooses which moves to play until a terminal state is reached.

Backpropagation: the result of the simulation is propagated back through all the nodes traversed in the tree.

When the search budget expires, MCTS returns the best move in the root node to be played in the real game. The best move might be 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 [2] (Upper Confidence bounds applied to Trees), that sees the problem of choosing an action a^* in a node s of the tree as a MAB and uses the UCB1 sampling strategy [25] to select the move to visit next:

$$a^* = \underset{a \in A(s)}{\operatorname{argmax}} \left\{ Q(s,a) + C \times \sqrt{\frac{\ln N(s)}{N(s,a)}} \right\} \quad . \qquad (1)$$

$A(s)$ is the set of legal moves in s, $Q(s,a)$ is the average result of all simulations in which move a has been selected

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TG.2018.2884768, IEEE Transactions on Games

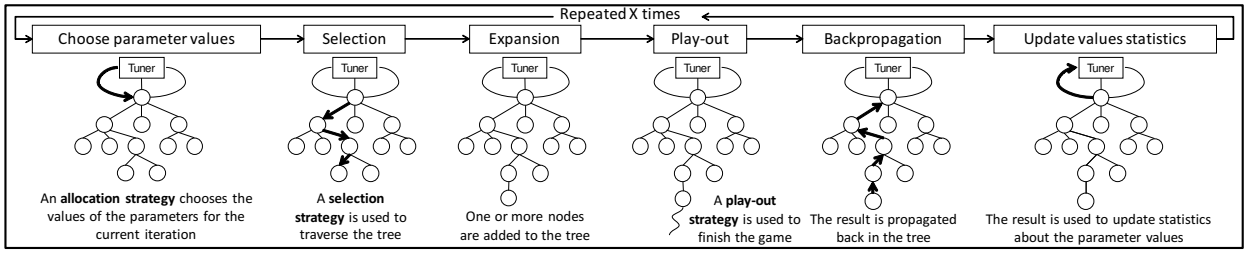

Fig. 1. Interleaving on-line tuning with MCTS (inspired by [28]).

in 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 move a has been selected whenever node s was visited. The C constant is used to control the balance between exploitation of good moves and exploration of less visited ones.

A selection strategy that proved successful in multiple domains, such as Knighttrough, Domineering, variants of Go and GGP is GRAVE [6], [7], a modification of the RAVE strategy [3], [5]. GRAVE selects a move according to (1), where the term $Q(s,a)$ is substituted by:

$$(1-\beta(s)) \times Q(s,a) + \beta(s) \times AMAF(s',a) \ . \quad (2)$$

Here, s' is the closest ancestor of s that has at least *Ref* visits (note that it might be s itself). The value $AMAF(s',a)$ is known as the *All Moves As First* [29] value. It represents the average result obtained from all simulations in which move a is performed at any time after node s' is visited. The AMAF values are used to increase the number of samples when selecting a move in nodes with a low number of visits. This way the variance of the move value estimates is reduced and the learning process is faster. The parameter $\beta(s)$ controls the importance of the AMAF value and decreases it over time, when the number of visits for the node increases. One of the proposed formulas to compute β is the following [3], [5]:

$$\beta(s) = \sqrt{\frac{K}{3 \times N(s) + K}} \ , \quad (3)$$

where K is the *equivalence parameter*, which indicates for how many simulations the two scores are weighted equal.

For the play-out phase, MAST [3] has shown to improve the performance over a simple random strategy. During the search, for each move a, MAST keeps track of a global average return value $Q_{MAST}(a)$ of all the simulations in which a was played. Then, when selecting a move for a certain game state in the play-out, it chooses the move with the highest $Q_{MAST}(a)$ with probability $(1-\epsilon)$ or a random move with probability ϵ.

IV. Design of self-adaptive MCTS

This section presents the two main aspects of the proposed SA-MCTS. Subsection IV-A discusses how the tuning strategy can be integrated within the MCTS algorithm to make it adaptive. Subsection IV-B presents the formulation of the tuning problem.

A. Integration of Parameter Tuning with MCTS

Figure 1 shows how parameter tuning is interleaved with MCTS simulations. For each iteration of the algorithm a *Tuner* uses an *allocation strategy* to choose a combination of values for the parameters. Next, the four phases of MCTS are performed using the selected parameter values to control the search. The result obtained by the simulation is used to update statistics about the quality of the chosen parameter combination. This process continues until the end of the game, such that the parameters are tuned on-line for the whole duration of the search.

B. Formulation of the Parameter Tuning Problem

An allocation strategy is required to decide how to divide the available number of samples among all the combinations of parameter values that have to be evaluated. An ideal allocation strategy for the on-line parameter tuning problem should aim to assign most of the samples to good value combinations, reducing the number of samples assigned to bad value combinations. This is because each evaluated combination has an impact on the quality of the actual search. If bad combinations are evaluated too often the quality of the search results will decrease.

The main idea behind the formulation of the tuning problem is based on the work presented in [23]. This article discusses multiple allocations strategies for a problem similar to ours (on-line adaptation of the search strategy to the played game). Among all the approaches they show that the one considering the simulation allocation as a MAB problem combined with UCB selection is the one that assigns the highest number of samples to the best search strategy and the lowest to the worst.

The action-space of the on-line parameter tuning problem has a combinatorial structure (i.e. the action of choosing a parameter setting consists of multiple sub-actions that assign a certain value to each of the parameters). For this reason, instead of considering the tuning problem as a MAB, this article considers it as a CMAB (either with discrete or continuous variables) and bases the design of the allocation strategies on this formulation.

The CMAB with discrete variables is defined by the following three components:

- Set of d variables, $P = \{P_1, ..., P_d\}$, where each variable P_i can take m_i different values $V_i = \{v_i^1, ..., v_i^{m_i}\}$.
- Reward distribution $R : V_1 \times ... \times V_d \rightarrow \mathbb{R}$ that depends on the combination of values assigned to the variables.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TG.2018.2884768, IEEE Transactions on Games

- Function $L : V_1 \times ... \times V_d \rightarrow \{true, false\}$ that determines which combinations of values are legal.

The CMAB with continuous variables is defined analogously, by considering that each variable P_i can take a value in the interval $I_i = [min_{P_i}, max_{P_i}]$. For the parameter tuning problem, the parameters are considered as the variables of the CMAB.

V. Allocation strategies

This section introduces five allocation strategies: Naïve Monte Carlo [9], [10] (NMC, Subsection V-A1), Linear Side Information [11] (LSI, Subsection V-A2), Evolutionary Algorithm (EA, Subsection V-A3), N-Tuple Bandit Evolutionary Algorithm [12], [13], [16] (NTBEA, Subsection V-A4), and Covariance Matrix Adaptation Evolutionary Strategy [14] (CMA-ES, Subsection V-B1). The first four consider a discrete domain for the parameter values, while the latter considers a continuous domain.

For the sake of simplicity, the pseudocode of the different allocation strategies is given for one-player games. When tuning parameters for two- or multi-player games, all the allocation strategies compute a different combination of parameters for each role in the game independently (i.e. each role has its own instance of the allocation strategy). All the computed combinations of parameters are then used to control the same MCTS simulation. In this way, parameter value combinations are co-evolved for all the roles. Note that having a different parameter combination for each role means that during the MCTS simulation different instantiations of selection or play-out strategies might be used to choose the actions for the different roles.

A. Discrete Allocation Strategies

This section presents the four allocation strategies that consider the tuning problem as a CMAB with a discrete domain. The first two strategies, NMC and LSI, have already been proposed by previous research to specifically deal with CMABs. EA and NTBEA, being based on evolutionary computation, have not been specifically designed for CMABs, but can still be applied in this context.

1) Naïve Monte-Carlo: This strategy was first proposed by Ontañón [9], [10] and applied to real-time strategy games (known for having a combinatorial action-space).

Algorithm 1 shows the pseudocode for NMC. The procedure NmcParameterTuning() implements the structure discussed in Subsect. IV-A. The procedure ChooseParameterValues() shows how NMC chooses the combination of parameter values to test before an MCTS simulation. Two main steps are distinguished, *exploration* that generates new parameter combinations, and *exploitation* that evaluates the combinations generated so far. These two steps are interleaved and for each iteration a policy π_0 chooses which one to perform. The *exploration* is based on the *naïve assumption*: $R(\vec{p} = \langle p_1, ..., p_d \rangle) \approx \sum_{i=1}^{d} R_i(p_i)$, where $\vec{p}$ is a vector representing a possible assignment of values $\langle p_1, ..., p_d \rangle$ to the parameters. This means that the expected reward of a certain configuration of parameter values can be approximated by a linear combination of expected rewards of single parameter values, as if considering the parameters to be independent. More precisely, the *exploration* considers d local MABs, one per parameter and uses them independently to generate a new combination of parameter values. Each local MAB has an arm for each possible value of the associated parameter. A policy π_l is used to select one value p_i for each parameter P_i using the corresponding local MAB (i.e. MAB_i). The resulting combination of values, if not yet present, is added to the global MAB (i.e. MAB_g) used during the *exploitation*. MAB_g considers each arm to be associated to a possible parameter combination. Initially it has no arms and is filled during *exploration*. The *evaluation* uses a policy π_g to select from MAB_g a parameter combination to evaluate.

Algorithm 1 Pseudocode for NMC

```
1: procedure NmcParameterTuning( )
2:     while game not over do
3:         p⃗ ← ChooseParameterValues( )
4:         r ← PerformMctsSimulation(p⃗)
5:         UpdateStatistics(p⃗, r)
1: procedure ChooseParameterValues( )
   Output: Combination of parameter values p⃗ = ⟨p_1, ..., p_d⟩.
2:     phase ← π_0.choosePhase( )
3:     if phase = exploration then                ▷ Generate combination
4:         for i ← 1, ..., d do
5:             p_i ← π_l.ChooseValue(MAB_i)
6:         MAB_g.Add(p⃗)
7:     else if phase = exploitation then          ▷ Eval. combination
8:         p⃗ ← π_g.ChooseCombination(MAB_g)
9:     return p⃗
1: procedure UpdateStatistics(p⃗, r)
   Input: Chosen parameter values p⃗, reward r obtained from the
   simulation controlled by parameter values p⃗.
2:     MAB_g.UpdateArmStatistics(p⃗, r)
3:     for i ← 1, ..., d do
4:         MAB_i.UpdateArmStatistics(p_i, r)
```

The procedure UpdateStatistics($\vec{p}, r$) shows how the reward of the MCTS simulation is used to update statistics about the chosen parameter values. Statistics are updated in the global MAB for the given combination and in the local MABs for each value in the combination.

2) Linear Side Information: The LSI algorithm [11] is similar to NMC and distinguishes two main steps, called *generation* and *evaluation*. The *generation* step, like the *exploration* step of NMC, generates new combinations of parameter values considering them as independent (i.e. making a *naïve assumption*), while the *evaluation* step, like the *exploitation* step of NMC, evaluates the generated combinations. The main difference with NMC is that LSI performs these two steps in sequence instead of interleaving them, and a total predefined budget of available samples $N = N_g + N_e$ is divided among them.

Algorithm 2 gives the pseudocode for LSI. The procedure LsiParameterTuning(N_g, N_e, k) implements the main logic of LSI. The *generation* uses up to N_g samples (i.e. MCTS simulations) to generate a set $C^* \subseteq C = V_1 \times ... \times V_d$ of at most k legal combinations of parameters. The *evaluation* uses up to N_e samples to evaluate the combinations of values in C^* and recommend the best one, $\vec{p}^*$. When both phases of LSI are over, the recommended best combination $\vec{p}^*$ is used to control the rest of the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TG.2018.2884768, IEEE Transactions on Games

Algorithm 2 Pseudocode for LSI

procedure LsiParameterTuning(N_g, N_e, k)
Input: #samples N_g for the *generation* phase, #samples N_e for the *evaluation* phase, #candidates k to generate.
  $C^* \leftarrow$ Generate(N_g, k)
  $\vec{p}^* \leftarrow$ Evaluate(C^*, N_e)
  while game not over **do**
    PerformMctsSimulation($\vec{p}^*$)

procedure Generate(N_g, k)
Input: #samples N_g for the *generation* phase, #candidates k to generate.
Output: Set of candidate parameter combinations to evaluate, C^*.
  $\hat{R} \leftarrow$ SideInfo(N_g)
  $C^* \leftarrow \emptyset$
  for k times **do**
    $\vec{p} = \langle p_1, \ldots, p_d \rangle \leftarrow$ empty array of size d
    $V \leftarrow \bigcup_{i=1}^{d} V_i$
    while $V \neq \emptyset$ **do**
      $v_i^j \sim \mathcal{D}[\hat{R} \restriction_V]$
      $V \leftarrow V \setminus V_i$
      $p_i \leftarrow v_i^j$
    $C^* \leftarrow C^* \cup \{\vec{p}\}$
  return C^*

procedure SideInfo(N_g)
Input: #samples N_g for the *generation* phase.
Output: Weight function $\hat{R}$ over single parameter values.
  $V \leftarrow \bigcup_{i=1}^{d} V_i$
  $x \leftarrow \left\lfloor \frac{N_g}{|V|} \right\rfloor$
  for x times **do**
    for each $v_i^j \in V$ **do**
      $\vec{p} \leftarrow$ RandomlyExtend(v_i^j)
      $r \leftarrow$ PerformMctsSimulation($\vec{p}$)
      average $\hat{R}(v_i^j)$ with r
  return $\hat{R}$

procedure Evaluate(C^*, N_e)
Input: Set of parameter combinations to evaluate C^*, #samples N_e for the *evaluation* phase.
Output: Best parameter combination.
  $C_0 \leftarrow C^*$
  for $i \leftarrow 0$ **to** $(\lceil \log_2 |C^*| \rceil - 1)$ **do**
    $x \leftarrow \left\lfloor \frac{N_e}{|C_i| \lceil \log_2 |C^*| \rceil} \right\rfloor$
    for x times **do**
      for each $\vec{p} \in C_i$ **do**
        $r \leftarrow$ PerformMctsSimulation($\vec{p}$)
        average expected value of $\vec{p}$ with r
    $C_{i+1} \leftarrow \lceil |C_i|/2 \rceil$ elements with highest estimated value
  return the only combination $\vec{p} \in C_{\lceil \log_2 |C^*| \rceil}$

MCTS simulations until the game terminates. The PerformMctsSimulation($\vec{p}$) procedure, before returning the control to the LSI procedure to continue the tuning, takes care of playing a move in the real game if the timeout for the current game step is reached.

The procedure SideInfo(N_g) constructs the function $\hat{R} : \bigcup_{i=1}^{d} V_i \to \mathbb{R}$, that associates to each parameter value v_i^j the average reward $\hat{R}(v_i^j)$ obtained by all the MCTS simulations that were allocated to v_i^j. To construct $\hat{R}$ the procedure divides equally over all the parameter values the total number of generation samples N_g. Each time a parameter value v_i^j is sampled using an MCTS simulation the other parameters are set to random values.

The procedure Generate(N_g, k) uses the function $\hat{R}$ to generate up to k combinations of parameter values. To do so, the function $\hat{R}$ is normalized to create a probability distribution over (a subset of) its domain. The notation $\mathcal{D}[\hat{R} \restriction_V]$ indicates the probability distribution induced by $\hat{R}$ over the subset V of its domain. Each combination is generated by repeatedly sampling a value from the distribution $\mathcal{D}[\hat{R} \restriction_V]$. The first time $V = \bigcup_{i=1}^{d} V_i$ (i.e. all the domain). For each subsequent step the set of available values V_i for the last set parameter P_i is removed from V.

The procedure Evaluate(C^*, N_e) uses *sequential halving* [30] to repeatedly evaluate the generated combinations and finally recommend one. *Sequential halving* performs multiple iterations dividing equally among them the available samples N_e. During each iteration the combinations are sampled uniformly and only half of them is kept for the next iteration (the half with the highest expected value). This process ends when only one combination is left.

It is important to note that LSI, as opposed to the other allocation strategies, is based on a fixed number of samples N that must be set in advance. Choosing a value for N is not trivial, if the value is too high the search is likely controlled by parameter values selected randomly, because the game might terminate before LSI reaches the *evaluation* step. On the contrary, if the value is too low the search is likely controlled by a sub-optimal combination, recommended using only a low number of samples. Ideally, N should correspond to the total number of available simulations for the game. Because in GGP is not possible to know this exact number in advance, N is estimated during the start-clock using the average game length of the performed simulations and the number of combinations that can be sampled per game step.

3) Evolutionary Algorithm: A subset of evolutionary computation is Evolutionary Algorithms (EAs) [26], which, at each generation, select a subset of individuals from the current population as elite, and reproduce the population using the elite by crossover and mutation. EAs do not rely on any assumption on the fitness function or fitness landscape. For the tuning problem, an EA can be seen as a budget allocation strategy which decides to spend more or less budget on some individuals, thus parameter settings of the agent. A combination of parameter values is considered as an individual, where each single parameter is a gene. The fitness of each individual is computed as the reward obtained by the MCTS simulation controlled by the corresponding parameter values.

Algorithm 3 shows the pseudocode for EA. The main algorithm is implemented by the procedure EaParameterTuning(λ, μ, p_{cross}). The initial population Λ of size λ is generated randomly. Until the game is over, the current population is evaluated using MCTS simulations and evolved. When evolving, the μ elite individuals of the population (i.e. the ones with highest fitness) are used to generate $\lambda - \mu$ new individuals. These new individuals, together with the elite will form the new population.

The procedure GenerateIndividual(M, p_{cross}) shows how a new individual is generated. With probability p_{cross} a new individual is generated by uniform crossover of two randomly selected elite individuals. Otherwise it is generated by random mutation of a single parameter value of a randomly selected elite individual.

4) N-Tuple Bandit Evolutionary Algorithm: Recently, a new type of Evolutionary Algorithm called N-Tuple Bandit Evolutionary Algorithm (NTBEA) has been proposed by Lucas *et al.* [13]. Kunanusont *et al.* [12] applied it to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TG.2018.2884768, IEEE Transactions on Games

Algorithm 3 Pseudocode for EA

procedure EAPARAMETERTUNING(λ, μ, p_{cross})
Input: Population size λ, elite size μ, probability of generating an individual by uniform crossover p_{cross}.
  $\Lambda \leftarrow \emptyset$ ▷ Empty population set
  for $i \leftarrow 1, ..., \lambda$ **do**
    $\vec{p} \leftarrow$ GENERATERANDOMINDIVIDUAL( )
    $\Lambda \leftarrow \Lambda \bigcup \{\vec{p}\}$
  while game not over **do**
    for $\vec{p} \in \Lambda$ **do**
      $r \leftarrow$ PERFORMMCTSSIMULATION($\vec{p}$)
      UPDATEFITNESS($\vec{p}$, r)
      if game over **then**
        return
    $M \leftarrow$ get μ individuals in Λ with highest fitness
    $\Lambda \leftarrow M$
    for $i \leftarrow \mu + 1, ..., \lambda$ **do**
      $\vec{p} \leftarrow$ GENERATEINDIVIDUAL(M, p_{cross})
      $\Lambda \leftarrow \Lambda \bigcup \{\vec{p}\}$

procedure GENERATEINDIVIDUAL(M, p_{cross})
Input: Set M of elite individuals in the population, probability of generating an individual by uniform crossover p_{cross}.
Output: The generated individual.
  if RAND$(0, 1) < p_{cross}$ **then**
    $parent_1 \leftarrow$ random individual in M
    $parent_2 \leftarrow$ random individual in M
    return UNIFORMCROSSOVER($parent_1, parent_2$)
  else
    $parent \leftarrow$ random individual in M
    return SINGLERANDOMMUTATION($parent$)

Algorithm 4 Pseudocode for *LModel*

procedure INIT($\mathcal{L}$, P)
Input: Set $\mathcal{L}$ with the length of the n-tuples that we want to consider, and set of parameters to tune P.
  $nTuples \leftarrow \emptyset$
  for $l \in \mathcal{L}$ **do**
    $lTuples \leftarrow$ generate from P all n-tuples of length l
    $nTuples \leftarrow nTuples \cup lTuples$
  for $t \in nTuples$ **do**
    $LUT_t \leftarrow \emptyset$

procedure UPDATESTATISTICS($\vec{p}, r$)
Require: Initialized set of n-tuples, $nTuples$.
Input: Combination of parameter values, $\vec{p}$, and reward r obtained from the simulation controlled by parameter values $\vec{p}$.
  for $t \in nTuples$ **do**
    if not $LUT_t.contains(\vec{p}|_t)$ **then**
      $LUT_t.put(\vec{p}|_t)$
    $entry \leftarrow LUT_t.get(\vec{p}|_t)$
    $entry.rsum \leftarrow entry.rsum + r$
    $entry.n \leftarrow entry.n + 1$
    $LUT_t.n \leftarrow LUT_t.n + 1$

procedure UCBVALUE($\vec{p}$, C_{NTBEA})
Require: Initialized set of n-tuples, $nTuples$.
Input: Parameter value combination $\vec{p}$ for which to compute the UCB value, exploration constant C_{NTBEA} for the UCB formula.
Output: UCB value of the given parameter combination.
  $UCB \leftarrow 0$
  $count \leftarrow 0$
  for $t \in nTuples$ **do**
    $entry \leftarrow LUT_t.get(\vec{p}|_t)$
    if $entry \neq null$ **then**
      $UCB_{\vec{p}|_t} \leftarrow \frac{entry.rsum}{entry.n} + C_{\text{NTBEA}} \times \sqrt{\frac{\ln(LUT_t.n)}{entry.n}}$
      $UCB \leftarrow UCB + UCB_{\vec{p}|_t}$
      $count \leftarrow count + 1$
  if $count > 0$ **then**
    return $\frac{UCB}{count}$
  else
    return 0

automatic game parameter tuning. Sironi *et al.* [16] used it for on-line tuning of search-control parameters of an MCTS agent in GVGP. Like the previously presented EA, NTBEA considers each combination of parameters as an individual that is evolved over time by mutating single parameter values. Three components can be identified for NTBEA, the main *Evolutionary Algorithm*, a *noisy fitness evaluator* (represented in our case by MCTS simulations that evaluate parameter combinations), and an *n-tuple bandit fitness landscape model* (*LModel*, Algorithm 4).

NTBEA uses *LModel* to memorize statistics (e.g. mean, standard deviation, number of evaluations) of every legal value of every parameter and uses this model in combination with a bandit approach to decide which of the individuals should be evaluated next. Similar to the NMC approach, beside modeling each parameter as a bandit and each value of a parameter as an arm, each tuple formed by a subset of the available parameters is modeled as a macro-arm (e.g. if all the 2-tuples of a d-dimensional problem are considered, then $\frac{d(d-1)}{2}$ macro-arms will be used).

Algorithm 4 gives the pseudocode for the implementation of *LModel*. Given a d-dimensional search space, this model sub-samples its dimensions with a number of n-tuples with lengths that can range from 1 to d. Not all lengths in the range $[1, d]$ must necessarily be considered, but a set $\mathcal{L} \subseteq \{1, \ldots, d\}$ with the lengths l that we want to use can be specified. The procedure INIT($\mathcal{L}, P$) shows how the landscape model is initialized. Given the set of parameters P, for each of the specified lengths $l \in \mathcal{L}$ all the possible l-tuples t are generated, and for each of them an empty look-up table LUT_t is created. For example, given $P = \{P_1, P_2, P_3\}$ and $\mathcal{L} = \{1, 2, 3\}$ the following n-tuples with their own LUT would be created: $\langle P_1 \rangle, \langle P_2 \rangle, \langle P_3 \rangle, \langle P_1, P_2 \rangle, \langle P_1, P_3 \rangle, \langle P_2, P_3 \rangle, \langle P_1, P_2, P_3 \rangle$.

Whenever a combination of parameter values $\vec{p}$ is evaluated by performing an MCTS simulation, the obtained reward r is used to update the LUT of each n-tuple as shown in the procedure UPDATESTATISTICS($\vec{p}, r$). The notation $\vec{p}|_t$ indicates the vector of values in $\vec{p}$ that are assigned to the parameters considered by the n-tuple t. For example, given the set of parameters $P = \{P_1, P_2, P_3\}$, the parameter combination $\vec{p} = \langle p_1, p_2, p_3 \rangle$, and the n-tuple $t = \langle P_1, P_3 \rangle$, we will have $\vec{p}|_{\langle P_1, P_3 \rangle} = \langle p_1, p_3 \rangle$. For each n-tuple t, the entry in LUT_t that corresponds to the value assignment $\vec{p}|_t$ is updated by increasing by 1 the number of visits and by r the total reward sum. Finally, the number of visits of LUT_t is also increased by 1.

The procedure UCBVALUE($\vec{p}$, C_{NTBEA}) computes the UCB value of a given combination of parameters $\vec{p}$ using *LModel*. First, for each vector of values $\vec{p}|_t$ with at least one visit we compute the UCB1 value $UCB_{\vec{p}|_t}$ considering the corresponding LUT_t as a MAB. This means that each arm of the MAB corresponds to an entry in LUT_t, and

Algorithm 5 Pseudocode for NTBEA.

procedure NTBEAPARAMETERTUNING(x, $\mathcal{L}$, P, C_{NTBEA})
Input: #neighbors x to generate during evolution, set $\mathcal{L}$ with the length of the n-tuples that we want to consider, set of parameters to tune P, exploration constant C_{NTBEA} to compute the UCB values.
  $LModel$.INIT($\mathcal{L}, P$)
  $\vec{p} \leftarrow$ GENERATERANDOMINDIVIDUAL( )
  while game not over **do**
    $r \leftarrow$ PERFORMMCTSSIMULATION($\vec{p}$)
    $LModel$.UPDATESTATISTICS($\vec{p}, r$)
    $\mathcal{N} \leftarrow$ generate x neighbors of $\vec{p}$ by single random mutation
    $\vec{p} \leftarrow \text{argmax}_{\vec{p}\,' \in \mathcal{N}}(LModel.\text{UCBVALUE}(\vec{p}\,', C_{\text{NTBEA}}))$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TG.2018.2884768, IEEE Transactions on Games

Algorithm 6 Pseudocode for CMA-ES

```
1: procedure CmaEsParameterTuning(x⃗, σ, α)
   Input: Initial point (distribution mean) x⃗ and initial standard deviation (i.e. step-size) σ for the CMA-ES strategy, factor α used to compute the fitness penalty for infeasible solutions.
2:     cma ← InitializeCmaEs(x⃗, σ)
3:     while not(cma.IsStopped( )) do
4:         Λ ← cma.SamplePopulation( )
5:         f⃗ ← new vector of size |Λ|
6:         for p⃗_j ∈ Λ do
7:             f_j ← ComputeFitness(p⃗_j)
8:         cma.UpdateDistribution(f⃗)
9:     p⃗ ← cma.GetMeanSolution( )
10:    f ← ComputeFitness(p⃗)
11:    cma.SetFitnessOfMean(f)
12:    p⃗* ← cma.GetBestSolution( )
13:    (p⃗*, penalty) ← RepairIndividual(p⃗*, α)
14:    p⃗* ← DenormalizeIndividual(p⃗*)
15:    while game not over do
16:        PerformMctsSimulation(p⃗*)

1: procedure ComputeFitness(p⃗)
   Input: Individual for which to compute the fitness p⃗.
   Output: The fitness value of the given individual.
2:     (p⃗, penalty) ← RepairIndividual(p⃗, α)
3:     p⃗ ← DenormalizeIndividual(p⃗)
4:     r ← PerformMctsSimulation(p⃗)
5:     return (100 − r + penalty)

1: procedure RepairIndividual(p⃗, α)
   Input: Individual p⃗ to be repaired if at least one of its values is infeasible (∉ [0, 1]).
   Output: The repaired individual with its penalty.
2:     p⃗′ ← new vector of size |p⃗|
3:     for p_i ∈ p⃗ do
4:         if p_i ∉ [0, 1] then
5:             if p_i < 0 then
6:                 p′_i ← 0
7:             else
8:                 p′_i ← 1
9:         else
10:            p′_i ← p_i
11:    penalty ← α ‖p⃗ − p⃗′‖
12:    return (p⃗′, penalty)

1: procedure DenormalizeIndividual(p⃗)
   Input: Individual p⃗ for which each value p_i ∈ p⃗ must be denormalized from [0, 1] to its interval of feasible values [min_{P_i}, max_{P_i}].
   Output: Individual with denormalized values.
2:     p⃗′ ← new vector of size |p⃗|
3:     for p_i ∈ p⃗ do
4:         p′_i ← min_{P_i} + p_i(max_{P_i} − min_{P_i})
5:     return p⃗′
```

thus to a possible assignment of values to the parameters considered by the n-tuple t. Then, all the $UCB_{\vec{p}|_t}$ values are averaged to obtain the UCB value for combination $\vec{p}$.

Algorithm 5 gives the pseudocode of the NTBEA allocation strategy and shows how it uses *LModel*. NTBEA starts with a randomly generated parameter combination $\vec{p}$, evaluates it with an MCTS simulation and uses the obtained reward r to update statistics in *LModel*. Then, it generates x neighbors of $\vec{p}$ using single random mutations and computes their UCB value using *LModel*. The neighbor with the highest UCB value becomes the currently considered solution and the procedure is repeated.

B. Continuous Allocation Strategy

This subsection describes how CMA-ES is used as allocation strategy that considers the tuning problem as a CMAB with a continuous domain.

1) Covariance Matrix Adaptation Evolutionary Strategy: A powerful evolutionary computation technique is CMA-ES [14], a second-order method using the covariance matrix estimated iteratively by finite differences. It has been proved to be efficient for optimizing non-linear non-convex problems in the continuous domain without a-priori domain knowledge, thus no knowledge of the fitness landscape or the gradient function is required.

For each generation $(g + 1)$, CMA-ES generates a population of λ new individuals $\vec{p}_j^{\,(g+1)}$ by sampling the multivariate normal distribution $\mathcal{N}(\vec{0}, \mathbf{C}^{(g)})$ as follows:

$$\vec{p}_j^{\,(g+1)} \sim \vec{x}^{\,(g)} + \sigma^{(g)} \mathcal{N}\left(\vec{0}, \mathbf{C}^{(g)}\right) \quad . \tag{4}$$

Here, $\vec{x}^{\,(g)}$, $\sigma^{(g)}$ and $\mathbf{C}^{(g)}$ are the mean value of the search distribution, the step-size and the covariance matrix at generation g, respectively. After computing the fitness of the population at generation $(g + 1)$, the highest ranked individuals are used to update $\vec{x}$, σ and $\mathbf{C}$ for the next generation. More details can be found in the tutorial [31].

As allocation strategy for the tuning problem we use an existing implementation of CMA-ES [1]. Algorithm 6 shows how it has been integrated in the code. The procedure CmaEsParameterTuning$(\vec{x}, \sigma, \alpha)$ shows how the strategy works. The variable *cma* refers to an instance of the CMA-ES algorithm initialized with the given start point $\vec{x}$ and step-size σ. Until CMA-ES meets one of its termination criteria, *cma*.SamplePopulation() samples a new population and *cma*.UpdateDistribution$(\vec{f})$ uses its fitness to update the distribution. Upon termination, the mean value of the distribution is evaluated and its fitness updated. The overall best solution (parameter combination) is used to control the rest of the search.

Note that the computation of the fitness of an individual shown in procedure ComputeFitness$(\vec{p})$ needs some precautions. First of all, CMA-ES minimizes the fitness function, while we want to maximize it, thus the fitness is computed as $(100 - r)$, where 100 is the maximum achievable reward in GGP and r is the reward obtained by the MCTS simulation controlled by the given parameter combination $\vec{p}$. Moreover, when tuning with CMA-ES we consider each parameter to be feasible in the interval $[0, 1]$ and the optimum is expected to be in the hyper-cube $[0, 1]^d$ (d number of parameters). This has two implications: (i) CMA-ES could still sample individuals with some values outside $[0, 1]$ and (ii) the actual interval of feasible values for the parameters might be different from $[0, 1]$. In the first case, whenever an individual is infeasible, we compute its fitness as the fitness of a repaired individual to which we add a penalty. This has been implemented according to the tutorial [31] and is shown in the procedure RepairIndividual$(\vec{p}, \alpha)$. In the second case, before evaluating a combination of parameters with an MCTS simulation, all the values are denormalized from $[0, 1]$ to their own interval of feasible values as shown in the procedure DenormalizeIndividual$(\vec{p})$.

VI. Empirical Evaluation

This section presents an empirical evaluation of on-line parameter tuning and a comparison of the discussed allo-

[1] Code and details available at cma.gforge.inria.fr.

cation strategies. The setup of the performed experiments is presented in Subsect. VI-A, while Subsects. VI-B, VI-C, VI-D, VI-E, VI-F and VI-G report the obtained results.

A. Setup

On-line parameter tuning has been implemented in the framework provided by the open source GGP-Base project [32]. It is used to tune the search parameters of the following two GGP agents[2]:

- **SP**: an MCTS agent that uses UCT as selection strategy and MAST as play-out strategy.
- **AP**: a more advanced MCTS agent that uses GRAVE as selection strategy and MAST as play-out strategy.

The purpose of using a more advanced agent is twofold. First of all, AP has more search-control parameters and enables the experiments to verify how the allocation strategies scale when the search space increases. Second, we can use it to verify how on-line parameter tuning performs with a more informed selection strategy. Table I reports all the tunable parameters used by either SP or AP. Their default values are obtained by tuning them off-line in sequence on the following set of games [33]: *3D Tic Tac Toe*, *Breakthrough*, *Knightthrough*, *Skirmish*, *Battle*, *Chinook*, *Chinese Checkers*. For K, *Ref* and T the continuous domain has been restricted to a value much smaller than infinity because after a certain threshold all values have more or less the same effect on the search.

Below are the settings of all the allocation strategies:
NMC: the policy π_0 is set to an ϵ-greedy strategy, which performs exploration with probability $\epsilon_0 = 0.75$ and exploitation with probability $(1-\epsilon_0) = 0.25$. These values are the same as in [9]. The policies π_l and π_g are both set to UCB1 with exploration constants $C_l = C_g = 1$, thus with high exploration. Experiments with smaller values of C showed a decrease in performance, suggesting that for the tuning problem exploration is fundamental because the best parameter combinations might change over time.
LSI: the total number of available samples N is estimated during start-clock and divided among the *generation* and *evaluation* steps as follows: $N_g = 0.75N$ and $N_e = 0.25N$ (this keeps the proportion between *generation* and *evaluation* the same as the proportion between *exploration* and *exploitation* in NMC). When tuning only two parameters the number of generated combinations k is set to 20, while when tuning more than two parameters $k = 2\,000$.
EA: the population size λ is set to 50 and the elite size μ is set to 25. The probability of generating a new individual by uniform crossover p_{cross} is set to 0.5. Smaller values for μ were tested, but resulted in a decreased performance.
NTBEA: the number of neighbors that are generated x is set to 5 (higher values showed a decrease in performance of the agents). The length considered to generate the n-tuples are set to $\mathcal{L} = \{1, d\}$, where d is the number of tuned parameters. Experiments considering all possible lengths for the n-tuple showed no improvement in performance and a decrease in simulations per second performed by the agents. The exploration constant C_{NTBEA} used to compute UCB1 values in *LModel* is set to 0.2.
CMA-ES: according to the suggestions in the tutorial [31], the initial point $\vec{x}$ is set to a random point in $[0, 1]^d$ and σ is set to 0.3. The value of α used to compute the penalty of infeasible individuals is set to 100. In addition, we disable all termination criteria regarding the fitness function, so that the optimization continues even if the minimum fitness is reached or if no significant change in fitness is observed. The motivation behind this choice is that the best parameter combination for MCTS might change over time, thus we want to keep exploring the search space. All other settings for CMA-ES are left to the default values (see tutorial [31]).

In addition, the last available version of CadiaPlayer[3] [3], the three-time champion of the GGP competition (in 2007, 2008 and 2012) is used in a series of experiments as a benchmark to compare the performance of the best on-line tuning agent with the one of the off-line tuned agent.

Note that in GGP it is assumed that agents cannot remember previously learned knowledge in-between games. This means that both the game tree built by MCTS and the parameter statistics collected by the allocation strategies will be reset before each new game run.

All agents are tested on a set of 14 heterogeneous games [33]: *3D Tic Tac Toe*, *Breakthrough*, *Knightthrough*, *Chinook*, *Chinese Checkers* with 3 players, *Checkers*, *Connect 5*, *Quad* (the version played on a 7×7 board), *Sheep and Wolf*, *Tic Tac Chess Checkers Four* (*TTCC4*) with 2 and 3 players, *Connect 4*, *Pentago* and *Reversi*. For each experiment, two agents at a time are matched against each other. For each game, all possible assignments of agents to the roles are considered, except the two configurations that assign the same agent to each role. All configurations are run the same number of times until at least 500 games have been played. Each match runs with $1s$ start- and play-clock, except for the experiments that involve CadiaPlayer. In these experiments CadiaPlayer uses $10s$ start- and play-clock while the other agents use $1s$ start- and play-clock, because our agents use a PropNet-based reasoner, and thus can perform a higher number of simulations per second. Experimental results always report the average win percentage of one of the two involved agents with a 95% confidence interval. The average win percentage of the agents for a game is computed by splitting 1 point among the agents that achieved the highest score and assigning 0 points to all other agents.

B. On-line Tuning for the SP Agent

This series of experiments evaluated the application of on-line tuning to the MCTS agent SP. Table II shows the results obtained by the agents that tune the parameters C and ϵ on-line with each of the presented allocation strategies against the off-line tuned SP agent. All the on-line

[2]Code of agents and tuning strategies available at github.com/ChiaraS/GGP-Project

[3]Version of 18-11-2012. Downloaded from http://cadia.ru.is/wiki/public:cadiaplayer:main

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TG.2018.2884768, IEEE Transactions on Games

TABLE I
PARAMETERS CONSIDERED IN THE EXPERIMENTS WITH THEIR DESCRIPTION, DEFAULT VALUE, DISCRETE DOMAIN AND CONTINUOUS DOMAIN

Param.	Description	Default value	Discrete domain	Continuous domain
C	Exploration constant for the UCB selection	0.2	$\{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$	$[0, 1]$
ϵ	Probability of selecting a random action with MAST	0.4	$\{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$	$[0, 1]$
K	*Equivalence parameter* of GRAVE (note that with $K = 0$ the selection strategy becomes pure UCT)	250	$\{0, 10, 50, 100, 250, 500, 750, 1\,000, 2\,000, \infty\}$	$[0, 2000]$
Ref	Visit threshold used by GRAVE to choose the ancestor from which to compute the AMAF values	50	$\{0, 50, 100, 250, 500, 1\,000, 10\,000, \infty\}$	$[0, 10000]$
VO	When selecting a move with UCT or GRAVE in state s, select a random move a among the ones with $Q(a, s) \in [max_a(Q(a, s)) - VO, max_a(Q(a, s))]$	0.01	$\{0.001, 0.005, 0.01, 0.015, 0.02, 0.025\}$	$[0, 0.025]$
T	During selection at node s, if s has less than T visits, use the play-out strategy to select the next action instead	0	$\{0, 5, 10, 20, 30, 40, 50, 100, 200, \infty\}$	$[0, 200]$

tuned agents reach at least the same overall performance of the off-line tuned agent, except $SP_{CMA\text{-}ES}$. Nevertheless, its overall performance is still very close to the one of SP. Only in *Knightthrough* and *Breakthrough* the performance of $SP_{CMA\text{-}ES}$ is much worse than the one of SP.

The agent that performs overall best is SP_{NTBEA}. Among all tuning agents this is the one that achieves the highest win rate in most of the games. The performance of the other agents, SP_{NMC}, SP_{LSI} and SP_{EA}, is also better than the one of SP in many of the games. Each of these agents achieves the best performance in one or two of the tested games. Particularly remarkable is the performance of SP_{LSI} in *Quad*. For this game this agent reaches a much higher win rate than the other agents (81.2%).

C. On-line Tuning for the AP Agent

This series of experiments evaluated the application of on-line tuning to the more advanced MCTS agent AP. Three series of experiments were performed, where the agents tune on-line two, four and six parameters, respectively. Table III shows the results obtained by each of the agents that use one of the allocation strategies against the agent that is tuned manually off-line.

When tuning two parameters, other than the combination K and *Ref*, we tried tuning the combination C and ϵ, but the latter achieved a worse performance. For this reason, only results for K and *Ref* are reported. Looking at the results in the first part of the table, for two tuned parameters all discrete allocation strategies achieve at least the performance of the off-line tuned agent. AP_{EA} reaches the highest overall win percentage. However, if compared with the results obtained by tuning the agent SP, in this case none of the allocation strategies seems to be superior to all the others. Each of them is the best in a few of the games. This suggests that it is more difficult to tune parameters for an agent that uses a more informed search strategy. Moreover, parameter values might have less influence than on a less informed search strategy.

Results in the second and the third part of the table show that for all tuning agents the overall performance decreases with the increase in number of tuned parameters. It might be that not all parameters have the same importance and by tuning them we are introducing noise in the process. Moreover, more parameters mean a larger

TABLE II
WIN PERCENTAGE OF ON-LINE TUNED SP AGENT WITH DIFFERENT ALLOCATION STRATEGIES AGAINST OFF-LINE TUNED SP AGENT

Game	SP_{NMC}	SP_{LSI}	SP_{EA}	SP_{NTBEA}	$SP_{CMA\text{-}ES}$
3DTicTacToe	42.9(±4.18)	**48.9**(±4.12)	43.6(±4.16)	48.0(±4.13)	42.9(±4.11)
Breakthrough	**61.0**(±4.28)	51.2(±4.39)	51.0(±4.39)	**61.0**(±4.28)	35.6(±4.20)
Knightthrough	48.0(±4.38)	35.2(±4.19)	40.0(±4.30)	**48.8**(±4.39)	20.8(±3.56)
Chinook	39.4(±3.41)	40.6(±3.50)	56.1(±3.58)	**65.7**(±3.37)	58.9(±3.51)
ChineseCheckers3	45.0(±4.35)	40.7(±4.29)	44.6(±4.34)	**46.8**(±4.36)	42.5(±4.32)
Checkers	69.4(±3.83)	47.6(±4.17)	70.9(±3.79)	**74.6**(±3.63)	48.7(±4.17)
Connect 5	39.6(±3.24)	45.9(±3.37)	45.6(±3.42)	**46.0**(±3.28)	42.0(±3.51)
Quad	37.5(±4.04)	**81.2**(±3.22)	50.9(±4.10)	43.8(±4.12)	58.0(±4.11)
SheepAndWolf	44.0(±4.36)	**52.2**(±4.38)	47.0(±4.38)	44.4(±4.36)	49.2(±4.39)
TTCC4 2P	69.0(±3.97)	60.9(±4.24)	70.9(±3.88)	**73.3**(±3.80)	57.1(±4.28)
TTCC4 3P	**48.2**(±4.28)	45.7(±4.23)	45.8(±4.26)	44.4(±4.20)	46.1(±4.26)
Connect 4	51.1(±4.21)	**79.7**(±3.36)	62.8(±4.06)	58.3(±4.11)	71.9(±3.77)
Pentago	63.0(±4.16)	63.2(±4.10)	66.2(±4.04)	**69.8**(±3.90)	63.5(±4.13)
Reversi	49.3(±4.32)	41.6(±4.23)	**55.5**(±4.31)	48.8(±4.32)	44.1(±4.29)
Avg Win%	50.5(±1.12)	52.5(±1.12)	53.6(±1.12)	**55.3**(±1.11)	48.7(±1.12)

TABLE III
WIN PERCENTAGE OF ON-LINE TUNED AP AGENT WITH DIFFERENT ALLOCATION STRATEGIES AGAINST OFF-LINE TUNED AP AGENT, FOR INCREASING NUMBERS OF TUNED PARAMETERS

Game	AP_{NMC}	AP_{LSI}	AP_{EA}	AP_{NTBEA}	$AP_{CMA\text{-}ES}$
Tuning K and *Ref*					
3DTicTacToe	47.2(±4.1)	42.0(±4.0)	52.0(±4.13)	46.0(±4.11)	41.5(±4.0)
Breakthrough	51.4(±4.39)	40.8(±4.31)	47.8(±4.38)	48.6(±4.39)	35.2(±4.19)
Knightthrough	58.0(±4.33)	49.6(±4.39)	46.2(±4.37)	46.8(±4.38)	47.2(±4.38)
Chinook	56.6(±4.04)	55.0(±4.11)	60.2(±3.98)	63.5(±3.95)	65.0(±3.88)
ChineseCheckers3	52.8(±4.36)	49.4(±4.37)	52.6(±4.36)	51.0(±4.37)	44.2(±4.34)
Checkers	44.2(±4.06)	48.2(±4.11)	47.8(±4.1)	47.6(±4.13)	41.8(±4.04)
Connect 5	44.6(±3.12)	49.2(±3.16)	46.2(±3.08)	45.7(±3.05)	42.9(±3.17)
Quad	57.0(±4.14)	64.3(±3.91)	60.4(±3.99)	60.1(±4.05)	52.2(±4.14)
SheepAndWolf	50.4(±4.39)	49.4(±4.39)	53.2(±4.38)	52.2(±4.38)	50.8(±4.39)
TTCC4 2P	49.2(±4.23)	52.0(±4.25)	52.9(±4.22)	51.5(±4.19)	44.4(±4.22)
TTCC4 3P	48.4(±4.21)	53.0(±4.21)	49.5(±4.26)	48.4(±4.26)	43.0(±4.22)
Connect 4	54.1(±4.12)	53.3(±4.14)	55.3(±4.15)	55.6(±4.18)	50.8(±4.12)
Pentago	54.1(±4.25)	48.5(±4.26)	56.2(±4.17)	55.3(±4.21)	51.9(±4.18)
Reversi	48.0(±4.32)	45.2(±4.28)	47.1(±4.28)	46.9(±4.33)	39.4(±4.23)
Avg Win%	51.1(±1.11)	50.0(±1.11)	52.0(±1.11)	51.4(±1.12)	46.4(±1.11)
Tuning K, *Ref*, C and ϵ					
3DTicTacToe	39.8(±4.05)	42.3(±4.11)	**48.7**(±4.15)	39.5(±4.08)	37.1(±4.00)
Breakthrough	**60.6**(±4.29)	37.2(±4.24)	59.8(±4.30)	55.2(±4.36)	18.2(±3.39)
Knightthrough	**74.2**(±3.84)	53.8(±4.37)	68.4(±4.08)	68.2(±4.09)	28.6(±3.96)
Chinook	36.7(±3.75)	24.8(±3.55)	**52.3**(±4.05)	51.0(±4.05)	48.9(±4.17)
ChineseCheckers3	36.9(±4.22)	36.1(±4.20)	39.5(±4.27)	**42.7**(±4.32)	40.3(±4.29)
Checkers	37.8(±3.91)	19.7(±3.28)	**42.4**(±4.00)	40.4(±4.06)	30.2(±3.84)
Connect 5	30.7(±3.11)	**40.3**(±3.27)	29.1(±2.95)	28.6(±3.07)	20.3(±2.68)
Quad	37.9(±4.04)	**75.6**(±3.56)	35.3(±3.98)	51.9(±4.21)	45.9(±4.09)
SheepAndWolf	45.2(±4.37)	**49.0**(±4.39)	47.6(±4.38)	44.8(±4.36)	47.4(±4.38)
TTCC4 2P	44.6(±4.25)	22.6(±3.60)	45.4(±4.25)	**49.7**(±4.20)	34.4(±4.08)
TTCC4 3P	40.4(±4.17)	**47.3**(±4.24)	43.2(±4.21)	43.1(±4.18)	45.4(±4.26)
Connect 4	42.9(±4.15)	**59.1**(±4.18)	50.1(±4.19)	46.8(±4.20)	49.4(±4.24)
Pentago	38.8(±4.07)	**48.8**(±4.22)	41.7(±4.16)	43.5(±4.15)	41.2(±4.16)
Reversi	39.8(±4.24)	27.1(±3.83)	42.9(±4.28)	**45.1**(±4.33)	34.9(±4.12)
Avg Win%	43.3(±1.11)	41.7(±1.11)	46.2(±1.12)	**46.5**(±1.12)	37.3(±1.09)
Tuning K *Ref*, C, ϵ, T and VO					
3DTicTacToe	27.3(±3.71)	28.3(±3.74)	35.2(±3.99)	**38.6**(±4.05)	36.1(±3.89)
Breakthrough	28.6(±3.96)	23.0(±3.69)	**37.4**(±4.25)	31.6(±4.08)	18.0(±3.37)
Knightthrough	**46.2**(±4.37)	28.4(±3.96)	45.6(±4.37)	50.2(±4.39)	39.2(±4.28)
Chinook	18.3(±3.21)	9.5(±2.23)	21.7(±3.40)	31.6(±3.94)	**48.6**(±4.22)
ChineseCheckers3	28.0(±3.92)	**41.1**(±4.30)	30.2(±4.01)	28.0(±3.92)	35.9(±4.19)
Checkers	17.7(±3.13)	19.3(±3.25)	17.2(±3.06)	**20.9**(±3.42)	19.6(±3.32)
Connect 5	24.6(±2.98)	**41.2**(±3.31)	25.5(±2.89)	33.2(±3.26)	37.6(±3.21)
Quad	16.2(±3.02)	**63.5**(±4.01)	15.2(±3.00)	17.2(±3.13)	19.0(±3.26)
SheepAndWolf	40.2(±4.30)	48.8(±4.39)	46.2(±4.37)	46.2(±4.37)	**49.4**(±4.39)
TTCC4 2P	26.2(±3.74)	17.9(±3.32)	28.7(±3.91)	**33.6**(±4.03)	25.0(±3.73)
TTCC4 3P	34.0(±4.04)	38.7(±4.18)	38.9(±4.12)	39.4(±4.15)	**40.6**(±4.23)
Connect 4	35.0(±3.97)	**53.8**(±4.15)	43.8(±4.14)	30.6(±3.89)	36.6(±4.05)
Pentago	35.2(±4.05)	40.2(±4.17)	42.3(±4.19)	42.1(±4.18)	**42.8**(±4.22)
Reversi	33.9(±4.08)	31.4(±4.01)	32.2(±4.02)	33.5(±4.07)	**34.2**(±4.11)
Avg Win%	29.4(±1.03)	**34.7**(±1.07)	32.9(±1.06)	34.0(±1.07)	34.5(±1.07)

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TG.2018.2884768, IEEE Transactions on Games

TABLE IV
VARIATION (%) OF VISITED NODES PER SECOND OF ON-LINE TUNED AP AGENTS WITH RESPECT TO OFF-LINE TUNED AP AGENT

Game	AP speed	AP_{NMC}	AP_{LSI}	AP_{EA}	AP_{NTBEA}	AP_{CMA-ES}
3DTicTacToe	58731	−18.4%	0.2%	−6.9%	−19.7%	−8.0%
Breakthrough	51089	−11.9%	−3.3%	−4.2%	−14.6%	−4.5%
Knightthrough	42691	−12.9%	−2.2%	−5.1%	−18.7%	−7.5%
Chinook	33817	−9.8%	−0.5%	−3.1%	−11.2%	−4.5%
ChineseCheckers3	106074	−27.9%	−1.8%	−1.4%	−20.1%	−5.9%
Checkers	29767	−1.4%	1.9%	1.8%	1.1%	0.9%
Connect5	37488	−10.7%	3.2%	−1.9%	−9.4%	−4.8%
Quad	42358	−14.1%	−2.6%	−4.9%	−15.2%	−6.5%
SheepAndWolf	51036	−22.0%	−1.1%	−1.3%	−8.6%	−4.0%
TTCC42P	26936	−6.5%	−1.9%	−2.2%	−6.2%	−3.1%
TTCC43P	23462	−15.7%	−2.0%	0.6%	−8.1%	−6.8%
Connect4	114623	−24.3%	−6.0%	−3.9%	−23.6%	−9.3%
Pentago	86132	−12.7%	0.1%	1.4%	−9.0%	−3.1%
Reversi	8533	−1.9%	−0.5%	0.2%	−0.5%	−1.6%

TABLE V
WIN PERCENTAGE OF SP_{NTBEA} AND AP_{NTBEA} AGAINST AGENTS THAT RANDOMIZE PARAMETER VALUES BEFORE EACH GAME RUN

Game	SP_{NTBEA}	AP_{NTBEA}		
	2 param.	2 param.	4 param.	6 param.
3DTicTacToe	58.4(±4.11)	59.3(±4.06)	65.4(±3.94)	64.0(±4.03)
Breakthrough	80.6(±3.47)	65.2(±4.18)	88.4(±2.81)	80.8(±3.46)
Knightthrough	80.6(±3.47)	55.0(±4.37)	87.2(±2.93)	83.6(±3.25)
Chinook	77.8(±2.83)	62.8(±3.91)	77.1(±3.43)	63.7(±4.04)
ChineseCheckers3	54.5(±4.35)	56.0(±4.34)	58.1(±4.31)	49.2(±4.37)
Checkers	78.9(±3.34)	59.1(±4.10)	74.7(±3.57)	57.6(±4.15)
Connect 5	62.4(±3.30)	58.5(±3.19)	46.5(±3.73)	54.5(±3.73)
Quad	41.4(±4.12)	60.6(±4.02)	54.6(±4.21)	35.5(±4.08)
SheepAndWolf	53.6(±4.38)	51.8(±4.38)	47.6(±4.38)	51.4(±4.39)
TTCC4 2P	77.1(±3.63)	62.8(±4.10)	78.8(±3.47)	71.3(±3.92)
TTCC4 3P	52.4(±4.24)	56.6(±4.24)	50.8(±4.29)	49.5(±4.30)
Connect 4	44.6(±4.20)	65.6(±4.02)	52.6(±4.23)	51.3(±4.27)
Pentago	60.3(±4.01)	61.1(±4.15)	55.4(±4.13)	57.9(±4.16)
Reversi	63.1(±4.15)	56.2(±4.27)	58.1(±4.28)	54.3(±4.32)
Avg Win%	63.2(±1.07)	59.3(±1.10)	63.9(±1.08)	58.9(±1.12)

search space, with fewer good parameter combinations. For this reason, it could be more difficult for the tuning agents to converge to an optimal combination and they keep evaluating sub-optimal ones. It might also be that, by the time they identify better parameter combinations, the AP agent has already an advantage in the game because it was making better decisions from the start due to already tuned parameters. In particular, AP_{CMA-ES} is the agent that loses the most in performance when increasing the number of parameters to four. This strategy uses a continuous parameter domain, so its search space increases much more than for the other strategies. Moreover, in GGP only a few different rewards can be obtained (e.g. win, draw, loss). This often causes a flat fitness landscape for CMA-ES, negatively influencing the optimization.

Despite the statistically significant worsening of the performance in most of the games, it still seems to be beneficial to tune four parameters for a few of them. For *Knightthrough* and *Breakthrough*, for example, AP_{NMC}, AP_{EA} and AP_{NTBEA} perform better than AP when tuning four parameters rather than only two. The performance of P_{LSI} also increases when tuning four parameters for *Quad*. A reason for this might be that for these games the fixed parameters of AP are not optimal, but it could also be that optimal values are changing during the search and on-line tuning detects this. To support this hypothesis, we noticed that for different game runs on-line tuning tends to focus on multiple parameter combinations instead of converging always to the same.

If we compare the tuning agents with each other, for four parameters AP_{EA} and AP_{NTBEA} show the overall best performance, while for six parameters AP_{LSI}, AP_{NTBEA} and AP_{CMA-ES} are the ones performing best, having a win percentage around 34.0%. Over all the experiments, AP_{NTBEA} seems to be the best performing agent.

An aspect worth investigating that might be influencing the performance of the on-line tuning agents is the impact of the overhead of selecting parameter values. Table IV gives as reference the speed (i.e. average median of number of visited nodes per second) of the off-line tuned AP agent. For the self-adaptive AP agents that tune four parameters the table reports the percentage of speed variation with respect to off-line tuned AP. For almost all games parameter tuning decreases the speed, especially for AP_{NMC} and AP_{NTBEA}. We also looked at the speed decrease for two and six parameters as well and noticed that for AP_{LSI}, AP_{EA} and AP_{CMA-ES} it is always on average around −0.5% to −4.9%, while for AP_{NMC} goes on average from −2.4% for two parameters to −25.3% for six, and for AP_{NTBEA} from −8.9% for two parameters to −12.4% for six. This can be explained by the frequency with which these two agents have to perform costly UCB1 evaluations to select parameters. A decrease in speed, however, does not seem to always imply a negative performance. By looking at the number of iterations per second of the agents we noticed that in many games on-line tuned AP agents can perform more iterations than the off-line tuned AP agent, and this might have a positive effect on the search. The explanation for the increase in iterations might be that the constantly changing search-control parameters cause the agents to explore different parts of the search space (with shorter paths) than the ones explored by AP. This might be happening for the games of *Quad* and *Connect 4* with AP_{LSI}. This agent for these games has an increase of 13% and 15% in number of iterations, respectively, and it shows a better performance than AP.

D. On-line Parameter Tuning Validation

This series of experiments is designed to verify if on-line parameter tuning has an advantage over fixed parameter values when such values are performing poorly. Given that NTBEA seems to be the allocation strategy that performs the best in previous experiments, we consider the SP_{NTBEA} agent tuning two parameters and the AP_{NTBEA} agents tuning two, four and six parameters. Each agent is matched against the corresponding non-tuning agent that, before each game run, sets its parameters to randomly chosen values among the available ones. Each of these non-tuning agents will randomize only the values of the parameters that the corresponding self-adaptive agent is tuning. Testing the tuning agents against all possible agents with fixed settings is too time consuming because of the large number of available parameter combinations. Therefore, randomization is used in these experiments to guarantee that many of the fixed parameter combinations used by the non-tuning agents will be performing poorly.

Results are shown in Table V. For almost all games, the self-adaptive agents have a significantly better performance than the agents that randomize parameter values

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TG.2018.2884768, IEEE Transactions on Games

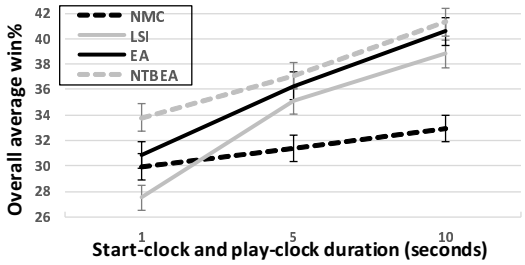

Fig. 2. Win percentage of AP_{NMC}, AP_{LSI}, AP_{EA} and AP_{NTBEA} tuning six parameters with different time constraints

TABLE VI
WIN PERCENTAGE OF AP AND AP_{NTBEA} (1s START- AND PLAY-CLOCK) AGAINST CADIAPLAYER (10s START- AND PLAY-CLOCK)

Game	AP	AP_{NTBEA}	
		2 parameters	4 parameters
3DTicTacToe	**92.1**(±2.36)	91.9(±2.34)	90.4(±2.55)
Breakthrough	63.2(±4.23)	61.8(±4.26)	**68.0**(±4.09)
Knightthrough	50.8(±4.39)	52.2(±4.38)	**74.8**(±3.81)
Chinook	82.8(±3.22)	**88.0**(±2.74)	81.3(±3.28)
Checkers	90.6(±2.32)	**91.2**(±2.28)	87.6(±2.71)
Connect 5	**70.4**(±3.18)	68.2(±3.29)	45.5(±3.78)
Quad	98.8(±0.96)	99.2(±0.78)	**99.4**(±0.68)
SheepAndWolf	56.8(±4.35)	**60.4**(±4.29)	51.6(±4.38)
Connect 4	68.2(±3.90)	**69.7**(±3.92)	63.2(±4.06)
Pentago	73.0(±3.80)	**78.1**(±3.52)	71.3(±3.80)
Avg Win%	74.7(±1.16)	**76.1**(±1.14)	73.3(±1.19)

before each game run, proving that on-line parameter tuning can converge to better parameter settings when the opponent's parameters are set to sub-optimal values.

E. Parameters Inter-dependency

All the proposed allocation strategies are designed to take into account that there is inter-dependency among the tuned parameters. As a validation of this inter-dependency assumption, we also tested the performance of an agent, AP_{LOCAL}, that tunes four parameters considering no inter-dependency. This agent chooses which combination to evaluate by selecting each parameter value using a separate MAB (i.e. like using only local MABs with NMC). The overall win percentage of this agent against AP reached only 39.1%(±1.09), lower than the win percentage of most of the tuning agents reported in Table III for four parameters. It was also observed that for most games at least one of the allocation strategies that exploit parameter inter-dependency can significantly outperform the strategy that does not. This can be seen as a confirmation that there is a dependency and it should be exploited.

F. Tuning Six Parameters with Different Time Constraints

This series of experiments matches the agents that tune on-line six parameters with a discrete parameter domain against AP for different time constraints. Figure 2 shows for each agent how the average win percentage over all the tested games changes when using longer start- and play-clock. Results for 1s differ from the ones presented in Table III because the experiment ran on a different server, where the agents could perform less simulations per second. With more time all agents increase their performance, even if none of them can reach the performance of the off-line tuned agent. Among the on-line tuning agents AP_{NMC} is the one that benefits the least from the increase in thinking time, while AP_{LSI} is the one that benefits the most. Overall, AP_{NTBEA} seems to be the best performing agent, though for 5s and 10s its confidence interval overlaps with the one of AP_{EA}.

G. Best On-Line Tuning Agent vs CADIAPLAYER

In this series of experiments the off-line tuned agent AP and the best on-line tuning agent AP_{NTBEA} are matched against CADIAPLAYER. For AP_{NTBEA} only the versions that tune two and four parameters were tested. Table VI shows the obtained results. Four games (*Chinese Checkers* with 3 players, *TTCC4* with 2 and 3 players, and *Reversi*) are excluded from the experiments because CADIAPLAYER encountered some errors while playing them. Results show that both AP and AP_{NTBEA} are better than CADIAPLAYER. For most of the games at least one of the two versions of AP_{NTBEA} performs better than AP. In line with previous results, AP_{NTBEA} that tunes four parameters performs overall worse than AP_{NTBEA} that tunes only two parameters, however their difference in performance is not very large. Overall, AP_{NTBEA} that tunes two parameters seems to be the one that performs the best against CADIAPLAYER.

VII. CONCLUSION AND FUTURE WORK

This article presented an on-line tuning method for search-control parameters that enables MCTS to be self-adaptive during game play. The performance of this method was evaluated in GGP. Five different allocation strategies were introduced and tested for parameter tuning: NMC, LSI, EA, NTBEA and CMA-ES. Results show that, when tuning two parameters, with any of the allocation strategies the agents can reach at least the same performance or surpass the off-line tuned agent. When tuning four parameters for the more advanced agent, the overall performance is lower than the off-line tuned agent, but still quite close. This is especially remarkable because only a single run of a game is used to tune the parameters, instead of a few hundred. On-line tuning of six parameters is much harder, especially when the thinking time is low. In addition, experiments show that the NTBEA on-line tuning agents have a better performance than agents with random fixed parameter settings. It may be concluded that the proposed approach is useful when off-line parameter tuning is infeasible, or in contexts like GGP, where parameters cannot be tuned in advance for each game, or when off-line tuning incurs in the risk of overfitting the values to the set of games selected for the purpose of tuning.

Comparing only the on-line tuning approaches, NTBEA has the best performance overall. It performs well on most of the games and for different numbers of tuned parameters. This is likely due to the fact that it merges the use of evolutionary computation with the multi-armed bandit approach. It also seems that a discrete domain for the parameters is sufficient to achieve a good performance. CMA-ES, which uses continuous domains, was not

performing better and, as mentioned in Section VI-C, its performance is influenced by the flat fitness landscape.

Future work could look into improving the CMA-ES allocation strategy by reformulating the fitness function and by including a restart strategy. It might also be interesting to see if other evolutionary strategies for continuous domains can perform well as allocation strategies. In addition, the fact that the self-adaptive agents are not able to choose which and how many parameters to tune is a limitation of this work. These choices can be seen as extra parameters of the agent and for this work their values were selected manually by off-line testing. Future work should design agents that consider these choices as part of the on-line automatic adaptation. Moreover, performing this decision on-line could help automatically reduce the size of the combinatorial search space by excluding less relevant parameters. Another possibility is to use SMAC [34], which builds explicit regression models to predict the performance of parameters. Finally, it would be interesting to see if the devised on-line parameter tuning method can be successfully applied to other domains as well.

Acknowledgment

This work was supported by the Netherlands Organisation for Scientific Research (NWO) in the framework of the GoGeneral Project (Grant No. 612.001.121), the Shenzhen Peacock Plan (Grant No. KQTD2016112514355531), the Science and Technology Innovation Committee Foundation of Shenzhen (Grant No. ZDSYS201703031748284), and the Program for University Key Laboratory of Guangdong Province (Grant No. 2017KSYS008).

References

[1] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *Comput. Games*, ser. LNCS, H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers, Eds., vol. 4630. Berlin, Germany: Springer, 2007, pp. 72–83.

[2] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *Proc. Eur. Conf. Mach. Learn.*, ser. LNCS, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds., vol. 4212. Berlin, Germany: Springer, 2006, pp. 282–293.

[3] H. Finnsson and Y. Björnsson, "Learning simulation control in General Game-Playing agents," in *Proc. 24th AAAI Conf. Artif. Intell.*, M. Fox and D. Poole, Eds., vol. 10, 2010, pp. 954–959.

[4] C. B. Browne *et al.*, "A survey of Monte Carlo tree search methods," *IEEE Trans. Comput. Intell. AI in Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012.

[5] S. Gelly and D. Silver, "Combining online and offline knowledge in UCT," in *Proc. 24th Int. Conf. Mach. Learn.*, Z. Ghahramani, Ed., 2007, pp. 273–280.

[6] T. Cazenave, "Generalized rapid action value estimation," in *Proc. 24th Int. Joint Conf. Artif. Intell.*, Q. Yang and M. Wooldridge, Eds., 2015, pp. 754–760.

[7] C. F. Sironi and M. H. M. Winands, "Comparison of rapid action value estimation variants for General Game Playing," in *2016 IEEE Conf. Comput. Intell. Games*, 2016, pp. 309–316.

[8] ——, "On-line parameter tuning for Monte-Carlo tree search in General Game Playing," in *Proc. Int. Joint Conf. Artif. Intell. Workshop Comput. Games*, 2018, pp. 75–95.

[9] S. Ontañón, "The combinatorial multi-armed bandit problem and its application to real-time strategy games," in *Proc. 9th Artif. Intell. Interactive Digital Entertainment Conf.* AAAI Press, 2013, pp. 58–64.

[10] ——, "Combinatorial multi-armed bandits for real-time strategy games," *J. Artif. Intell. Res.*, vol. 58, pp. 665–702, Mar. 2017.

[11] A. Shleyfman, A. Komenda, and C. Domshlak, "On combinatorial actions and CMABs with linear side information," in *Proc. 21st Eur. Conf. Artif. Intell.* IOS Press, 2014, pp. 825–830.

[12] K. Kunanusont, R. D. Gaina, J. Liu, D. Perez-Liebana, and S. M. Lucas, "The N-tuple bandit evolutionary algorithm for automatic game improvement," in *Congr. Evol. Comput.* IEEE, 2017, pp. 2201–2208.

[13] S. M. Lucas, J. Liu, and D. Perez-Liebana, "The N-tuple bandit evolutionary algorithm for game agent optimisation," in *Congr. Evol. Comput.* IEEE, 2018.

[14] N. Hansen, S. D. Müller, and P. Koumoutsakos, "Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES)," *Evol. Comput.*, vol. 11, no. 1, pp. 1–18, 2003.

[15] J. Levine *et al.*, "General Video Game Playing," in *Artif. Comput. Intell. in Games*, ser. Dagstuhl Follow-Ups, S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius, Eds. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2013, vol. 6, pp. 77–83.

[16] C. F. Sironi *et al.*, "Self-adaptive MCTS for General Video Game Playing," in *21st Int. Conf. Appl. Evol. Comput.*, ser. LNCS, K. Sim and P. Kaufmann, Eds., vol. 10784. Berlin, Germany: Springer, 2018, pp. 358–375.

[17] Y. Björnsson and T. A. Marsland, "Learning extension parameters in game-tree search," *Inform. Sci.*, vol. 154, no. 3, pp. 95–118, Sep. 2003.

[18] L. Kocsis, C. Szepesvári, and M. H. M. Winands, "RSPSA: Enhanced parameter optimization in games," in *Advances in Comput. Games*, ser. LNCS, H. J. van den Herik, S.-C. Hsu, T.-S. Hsu, and H. Donkers, Eds., vol. 4250. Berlin, Germany: Springer, 2006, pp. 39–56.

[19] G. M. J.-B. Chaslot, M. H. M. Winands, I. Szita, and H. J. van den Herik, "Cross-entropy for Monte-Carlo tree search," *Int. Comput. Games Assoc. J.*, vol. 31, no. 3, pp. 145–156, 2008.

[20] E. K. Burke *et al.*, "Hyper-heuristics: A survey of the state of the art," *J. Oper. Res. Soc.*, vol. 64, no. 12, pp. 1695–1724, Dec. 2013.

[21] A. Mendes, J. Togelius, and A. Nealen, "Hyper-heuristic general video game playing," in *2016 IEEE Conf. Comput. Intell. Games*, 2016, pp. 94–101.

[22] D. Perez-Liebana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius, and S. M. Lucas, "General Video Game AI: a multi-track framework for evaluating agents, games and content generation algorithms," *arXiv preprint arXiv:1802.10363*, 2018.

[23] M. Świechowski and J. Mańdziuk, "Self-adaptation of playing strategies in General Game Playing," *IEEE Trans. Comput. Intell. AI in Games*, vol. 6, no. 4, pp. 367–381, Dec. 2014.

[24] B. Bischl *et al.*, "Aslib: A benchmark library for algorithm selection," *Artif. Intell.*, vol. 237, pp. 41–58, Aug. 2016.

[25] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the Multiarmed Bandit problem," *Mach. Learn.*, vol. 47, no. 2-3, pp. 235–256, May 2002.

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

[27] J. Liu, J. Togelius, D. Pérez-Liébana, and S. M. Lucas, "Evolving game skill-depth using general video game AI agents," in *Congr. Evol. Comput.* IEEE, 2017, pp. 2299–2307.

[28] G. M. J.-B. Chaslot, M. H. M. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk, and B. Bouzy, "Progressive strategies for Monte-Carlo tree search," *New Math. Natural Comput.*, vol. 4, no. 3, pp. 343–357, 2008.

[29] B. Brügmann, "Monte Carlo Go," Max Planck Inst. of Physics, München, Germany, Tech. Rep., 1993.

[30] Z. Karnin, T. Koren, and O. Somekh, "Almost optimal exploration in multi-armed bandits," in *Proc. 30th Int. Conf. Mach. Learn.*, S. Dasgupta and D. McAllester, Eds., 2013, pp. 1238–1246.

[31] N. Hansen, "The CMA evolution strategy: A tutorial," *arXiv preprint arXiv:1604.00772*, 2016.

[32] S. Schreiber and A. Landau, "The General Game Playing base package," https://github.com/ggp-org/ggp-base, 2017.

[33] S. Schreiber, "Games - base repository," http://games.ggp.org/base/, 2017.

[34] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Int. Conf. Learn. Intell. Optim.* Springer, 2011, pp. 507–523.