On-line Parameter Tuning for Monte-Carlo Tree Search in General Game Playing

Chiara F. Sironi(✉) and Mark H. M. Winands

Games & AI Group, Department of Data Science and Knowledge Engineering
Maastricht University, Maastricht, The Netherlands
{c.sironi,m.winands}@maastrichtuniversity.nl

Abstract. Many enhancements have been proposed for Monte-Carlo Tree Search (MCTS). Some of them have been applied successfully in the context of General Game Playing (GGP). MCTS and its enhancements are usually controlled by multiple parameters that require extensive and time-consuming computation to be tuned in advance. Moreover, in GGP optimal parameter values may vary depending on the considered game. This paper proposes a method to automatically tune search-control parameters on-line for GGP. This method considers the tuning problem as a Combinatorial Multi-Armed Bandit (CMAB). Four strategies designed to deal with CMABs are evaluated for this particular problem. Experiments show that on-line tuning in GGP almost reaches the same performance as off-line tuning. It can be considered as a valid alternative for domains where off-line parameter tuning is costly or infeasible.

1 Introduction

Monte-Carlo Tree Search (MCTS) [13, 21] is a simulation-based search technique that has become popular in game playing and has found many application domains [6]. A domain where MCTS has seen particular success is General Game Playing [3]. GGP aims at creating agents that can play any abstract game by only being given its rules and without using prior knowledge. Moreover, in this domain the time available to select which moves to play is usually limited to a few seconds. What makes MCTS suitable for GGP is that (1) it does not necessarily require game-specific knowledge, (2) it favors the exploration of the most promising regions of the search space and (3) it can make a decision within any budget constraint.

Many search control strategies and enhancements have been proposed for MCTS in various domains [6]. Some of the strategies that had particular success are the Rapid Action Value Estimation (RAVE) technique [16, 18], its generalization, GRAVE [9], Progressive History [26], the Move Average Sampling Technique (MAST) [15] and its variant, the N-gram Selection Technique (NST) [36]. These strategies improve different phases of the search by exploiting in different ways information about the general performance of the moves (collected either in the whole game or in relevant sub-parts of the game tree).

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. Usually, extensive off-line tuning is required to find the best value for each parameter. 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 the best values for strategy control parameters are mainly game dependent [9, 26, 34, 36] and it is difficult to find a single set of values that works best for all games. This means that to achieve the best performance parameter values should be tuned for each game.

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 by devising an on-line tuning strategy that adjusts the parameter values for each new game being played. Such strategy should also aim at tuning the parameters in combination, because parameter values are usually interdependent.

This paper considers the problem of tuning on-line a finite set of search-control parameters for an MCTS-based GGP agent. Each parameter can assume one single value at a time from a predefined finite set of values. This results in a finite (usually high) number of possible values combinations that must be evaluated during the search.

The proposed approach interleaves parameter tuning with MCTS and uses each MCTS simulation to evaluate a different combination of parameter values. An allocation strategy is required to decide how many simulations must be used to evaluate each combination of values. This paper presents and evaluates four possible allocation strategies: Multi-Armed Bandit (MAB) allocation [1, 27], Hierarchical Expansion (HE) [30], Naïve Monte-Carlo (NMC) [27, 28] and Linear Side Information (LSI) [33].

The remainder of this paper is structured as follows. Section 2 introduces previous work related to parameter tuning. Section 3 gives background on the MAB problem and MCTS. The general structure of the on-line parameter tuning problem and the four proposed allocation strategies are discussed in Sect. 4. Results obtained by testing these strategies in the context of GGP are presented in Sect. 5, and Sect. 6 gives the conclusions and outlines possible future work.

2 Related Work

As mentioned in [17], in the research area of game playing, not much work has been performed on on-line learning of search control. One example of such work is [4]. This paper presents both an off-line and an on-line approach for learning search-extension parameters for $\alpha\beta$-search. The method is based on gradient descent and looks for the parameter values that minimize the growth rate (in the number of visited nodes) of the search.

More attention has been given to automated off-line tuning of search-control parameters. An example is the work of Kocsis *et al.* [20] that uses an enhanced version of the Simultaneous Perturbation Stochastic Approximation (SPSA) algorithm to tune parameters for Poker and Lines of Action. Another approach, that recalls the structure of evolutionary approaches, is presented in [10] and proposes to tune parameters for the game of Go using the Cross-Entropy Method (CEM). This method keeps a probability distribution over possibly good parameter values and uses the evaluation of samples drawn from it to refine the distribution over time. A genetic algorithm is the solution proposed in [12] to tune the parameters that control the behavior of a rule-based bot for a first-person shooter game.

What all these approaches have in common is that they require a high number of samples against a benchmark player in order to find an optimal parameter configuration. The final obtained configuration is then evaluated by matching an agent that uses it against one that uses manually tuned parameters. All papers conclude that automated tuning is at least equal to manual tuning.

CLOP [14] is another algorithm proposed for tuning game parameters and is based on local quadratic regression. As the other mentioned approaches, it also requires a fair amount of samples in order to find a good solution.

Another research area to mention is the one that focuses on designing Hyper-Heuristics. As defined in [8], a hyper-heuristic is *"a search method or learning mechanism for selecting or generating heuristics to solve computational search problems"*. A recent application of the Hyper-Heuristic concept is presented in [25]. This paper discusses the implementation of a hyper-agent for the General Video Game Playing [23] framework (GVG-AI). This agent uses a hyper-heuristic to select from a portfolio of sub-agents the best one for the game at hand. The approach works off-line by training the agent to recognize which controllers perform best depending on certain game features. On the contrary, the hyper-heuristic approach presented in [35] devises an on-line mechanism to select from a portfolio of strategies the one that is best suited for the current game.

The work proposed in our paper is somewhat similar to the concept of hyper-heuristic. Some of the parameters that can be tuned can decide whether to (de)activate a certain search-control strategy depending on the value that is assigned to them. In this sense, tuning the parameters can be seen as a hyper-heuristic to choose which strategies to apply from a portfolio of available strategies (determined by the available parameter configurations).

3 Background

This section provides background on the MAB problem (Subsect. 3.1) and on MCTS (Subsect. 3.2).

3.1 Multi-Armed Bandit

The MAB problem [1] with n arms is defined as a set of n unknown independent real reward distributions $R = \{R_1, ..., R_n\}$, each of which is associated to one of the arms. When one of the arms is played a reward is obtained as a sample of the corresponding distribution.

The aim of a sampling strategy for a MAB problem 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.

3.2 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 [13, 21]. 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: after reaching a terminal state, 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 that has 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 [21] (Upper Confidence bounds applied to Trees). UCT sees the problem of choosing an action in a certain node of the tree as a MAB problem and uses the UCB1 [1] sampling strategy to select the move to visit next. UCT selects in node s the action a that maximizes the following formula:

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

where $Q(s,a)$ is the average result obtained from all the simulations in which move a was played in node 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, some variants of Go and GGP is GRAVE [9, 34], a modification of the RAVE strategy [16, 18]. GRAVE selects the move that maximizes the UCB1 formula (1) where the term $Q(s,a)$ is substituted by:

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

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* [5, 7] value, and represents the average result obtained from all the simulations in which move a is performed at any moment after node s' is visited. The AMAF values are used to increase the number of samples when selecting a move in nodes that have a low number of visits. In 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 [16, 18]:

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

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

For the play-out phase, MAST [15] and its variant, NST [36] have 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. NST uses the same strategy as MAST, but keeps track of a global average return value for sequences of moves instead of just single moves.

4 On-line Parameter Tuning

This section presents the two main aspects of the proposed tuning strategy. Subsection 4.1 discusses how the tuning strategy can be integrated within the MCTS algorithm, while Subsect. 4.2 presents four allocation strategies that decide how to distribute the available samples among the different combinations of parameter values to be evaluated.

4.1 Integration of Parameter Tuning with MCTS

Figure 1 shows how parameter tuning is interleaved with MCTS simulations. First, for each iteration of the algorithm an *allocation strategy* chooses a combination of values for the parameters. Next, the four phases of MCTS described in Subsect. 3.2 are performed using the selected parameter values to control the search. Finally, the result obtained by the simulation is used to update statistics about the quality of the chosen combination of parameter values.

4.2 Allocation Strategies

An allocation strategy is required to decide how to divide the available number of samples among all the combinations of parameter values that must be evaluated. An ideal allocation strategy for the on-line parameter tuning problem should try to assign the highest number of samples to the optimal combination, reducing

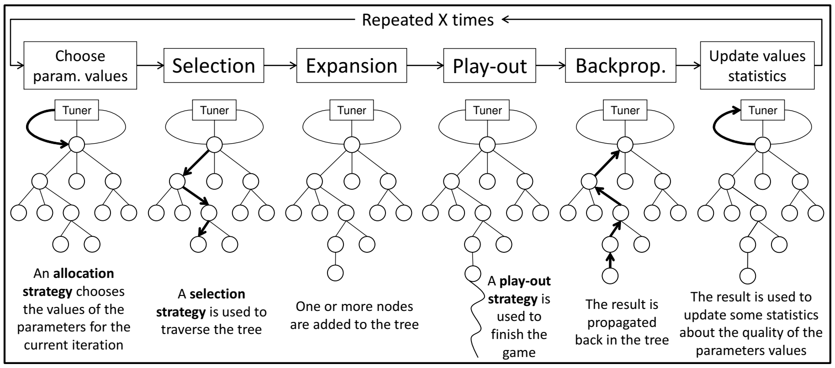
Repeated X times
Choose param. values
Selection
Expansion
Play-out
Backprop.
Update values statistics
Tuner
An **allocation strategy** chooses the values of the parameters for the current iteration
A **selection strategy** is used to traverse the tree
One or more nodes are added to the tree
A **play-out strategy** is used to finish the game
The result is propagated back in the tree
The result is used to update some statistics about the quality of the parameters values

Fig. 1. Interleaving on-line tuning with MCTS (Inspired by the figure representing the *Outline of a Monte-Carlo Tree Search* in [11])

to the minimum the number of samples assigned to bad values 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 design of the proposed allocation strategies is based on the work presented in [35]. This paper 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 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 allocation problem as a MAB, this paper considers it as a CMAB problem.

The CMAB problem is introduced in [27] as a variation of the MAB problem and it is used to represent decision problems for which the rewards depend on combinations of actions instead of single actions.

Following the definition of the CMAB problems, the problem of tuning multiple parameters simultaneously can be defined by the following three components:

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

Below four allocation strategies are introduced for the on-line parameter tuning problem. All of them have already been proposed by previous research to

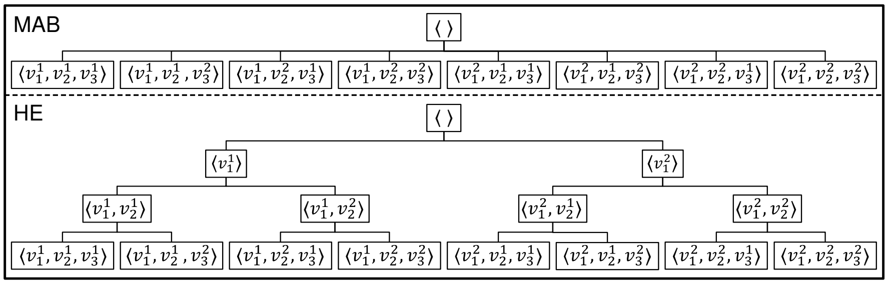
MAB
⟨ ⟩
⟨v11,v21,v31⟩ ⟨v11,v21,v32⟩ ⟨v11,v22,v31⟩ ⟨v11,v22,v32⟩ ⟨v12,v21,v31⟩ ⟨v12,v21,v32⟩ ⟨v12,v22,v31⟩ ⟨v12,v22,v32⟩
HE
⟨ ⟩
⟨v11⟩ ⟨v12⟩
⟨v11,v21⟩ ⟨v11,v22⟩ ⟨v12,v21⟩ ⟨v12,v22⟩
⟨v11,v21,v31⟩ ⟨v11,v21,v32⟩ ⟨v11,v22,v31⟩ ⟨v11,v22,v32⟩ ⟨v12,v21,v31⟩ ⟨v12,v21,v32⟩ ⟨v12,v22,v31⟩ ⟨v12,v22,v32⟩

Fig. 2. MAB and HE representation of the combinatorial action-space of the parameter tuning problem

deal with games that have a combinatorial action-space [27, 28, 30, 33]. For these games the problem of choosing an action in a state can be seen as a CMAB.

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 arrays of parameters are then used to control the MCTS strategy. Having a different parameter combination for each role means that during the same MCTS simulation different selection or play-out strategies might be used for different roles.

Multi-Armed Bandit Allocation. One trivial solution for dealing with a CMAB problem is to translate it back to a MAB [1, 27], where each arm corresponds to a possible legal combination of values for the parameters. Then the allocation strategy can use a policy π_{mab} to select the next combination to sample from the MAB problem.

In this way, however, the information on the combinatorial structure of the parameter values is lost. Often, a value that is good (or bad) for a parameter in a certain combination of values, is also good (or bad) in general or in many other combinations. With a MAB this information is ignored and cannot be exploited.

Hierarchical Expansion. As an alternative to the previously presented MAB representation for combinatorial action-spaces Hierarchical Expansion [30] is considered. The main idea behind HE is to represent the combinatorial action-space as a tree, where each level corresponds to a different parameter. In this way, the depth of the search of a combination of parameters is increased, but the branching factor is reduced.

Figure 2 shows how the problem of tuning 3 parameters, each with 2 possible values, is represented both with a MAB and with HE. To build the HE tree an order must first be imposed on the parameters. Each node of the tree corresponds to a partial combination of parameter values (the root corresponds to the empty combination). At every level of the tree the partial combination is extended by assigning a value to the next parameter in the order, until the combination is complete. The HE tree is then used to sample a combination of parameters for

```
procedure HeParametersTuning( )
    heTree ← BuildHeTree([$V_1, ..., V_n$])
    while game not over do
        $\boldsymbol{p}$ ← ChooseParameterValues(heTree)
        $r$ ← PerformMctsSimulation($\boldsymbol{p}$)
        UpdateValuesStatistics(heTree, $\boldsymbol{p}$, $r$)

procedure ChooseParameterValues(heTree)
Input: The HE tree, heTree.
Output: Combination of parameter values $\boldsymbol{p}$.
    node ← heTree.GetRoot( )
    $i$ ← 1
    while node corresponds to incomplete combination do
        $\boldsymbol{p}[i]$ ← $\pi_{he}$.ChooseValue(node)
        node ← heTree.GetNextNode(node, $\boldsymbol{p}[i]$)
        $i$ ← $i + 1$
    return $\boldsymbol{p}$

procedure UpdateValuesStatistics(heTree, $\boldsymbol{p}$, $r$)
Input: The HE tree, heTree. Chosen parameter values $\boldsymbol{p}$.
       Reward $r$ obtained from the simulation controlled
       by parameter values $\boldsymbol{p}$.
    node ← heTree.GetRoot( )
    $i$ ← 1
    while node corresponds to incomplete combination do
        node.UpdateArmStatistics($\boldsymbol{p}[i]$, $r$)
        node ← heTree.GetNextNode(node, $\boldsymbol{p}[i]$)
        $i$ ← $i + 1$
```

Fig. 3. Pseudocode for HE

each game simulation. MCTS can be applied to the HE tree, reducing to a MAB problem the choice of the next value to add to the combination.

Figure 3 gives the pseudocode for the HE allocation algorithm. The procedure HeParametersTuning() shows the main structure of the algorithm. First of all, the HE tree is built using the available parameter values V_i for each parameter P_i. The rest of the algorithm reflects the structure discussed in Subsect. 4.1, a combinations of parameters $\boldsymbol{p}$ is chosen and used to perform an MCTS simulation. The result of the simulation is then used to update statistics for the combination of parameters. The procedure PerformMctsSimulation($\boldsymbol{p}$) runs an MCTS simulation with the parameters $\boldsymbol{p}$ and implements all the four MCTS phases. It also takes care of checking the search budget for each game step and plays a move in the real game when this budget expires. This procedure is the same for all the allocation mechanisms discussed in this section.

The procedure ChooseParameterValues($heTree$) constructs a parameter combination $\boldsymbol{p}$ by visiting the HE tree. Starting from the root, the procedure visits one path in the tree. In each visited node a policy π_{he} is used to select the value for the next parameter and this value is added to the partial combination computed so far. The procedure UpdateValuesStatistics($heTree, \boldsymbol{p}, r$) propagates in the HE tree the reward r of the game simulation controlled by $\boldsymbol{p}$.

Naïve Monte-Carlo. First proposed in [27, 28], NMC is designed to exploit the combinatorial structure of the decision space and is based on the so called *naïve assumption*. For the parameter tuning problem this assumption can be expressed as follows:

$$R(\boldsymbol{p} = \langle p_1, ..., p_n \rangle) \approx \sum_{i=1}^{n} R_i(p_i) \ \ , \tag{4}$$

where, $\boldsymbol{p}$ is a vector representing a possible assignment of values $\langle p_1, ..., p_n \rangle$ 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.

```
procedure NmcParametersTuning( )
    while game not over do
        $\boldsymbol{p} \leftarrow$ ChooseParameterValues( )
        $r \leftarrow$ PerformMctsSimulation($\boldsymbol{p}$)
        UpdateValuesStatistics($\boldsymbol{p}, r$)

procedure UpdateValuesStatistics($\boldsymbol{p}, r$)
Input: Chosen parameter values $\boldsymbol{p}$.
       Reward $r$ obtained from the simulation
       controlled by parameter values $\boldsymbol{p}$.
    $MAB_g$.UpdateArmStatistics($\boldsymbol{p}, r$)
    for $i \leftarrow 1 : n$ do
        $MAB_i$.UpdateArmStatistics($\boldsymbol{p}[i], r$)

procedure ChooseParameterValues( )
Output: Combination of parameter values $\boldsymbol{p}$.
    $phase \leftarrow \pi_0$.choosePhase( )
    if $phase = exploration$ then                ▷ Generate combination
        for $i \leftarrow 1 : n$ do
            $\boldsymbol{p}[i] \leftarrow \pi_l$.ChooseValue($MAB_i$)
        $MAB_g$.Add($\boldsymbol{p}$)
    else if $phase = exploitation$ then          ▷ Eval. combination
        $\boldsymbol{p} \leftarrow \pi_g$.ChooseCombination($MAB_g$)
    return $\boldsymbol{p}$
```

Fig. 4. Pseudocode for NMC

The pseudocode for the NMC algorithm is shown in Fig. 4. The procedure NmcParametersTuning() implements the structure discussed in Subsect. 4.1.

The procedure ChooseParameterValues() shows how NMC chooses the combination of parameter values to test before an MCTS simulation. Two main phases are distinguished, an *exploration* phase that generates new parameter combinations and an *exploitation* phase that evaluates the combinations generated so far. These two phases are interleaved and for each iteration a policy π_0 chooses which of the two to perform.

The *exploration* phase considers n 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 also added to the global MAB (i.e. MAB_g) used by the *exploitation* phase. The global MAB considers each arm to be associated to a possible parameter combination. Initially MAB_g has no arms and is filled during the *exploration* phase. The *evaluation* phase uses a policy π_g to select form MAB_g an already generated parameter combination to evaluate.

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

Linear Side Information. The LSI algorithm [33] is similar to NMC and is based on the same *naïve assumption*. Like NMC, LSI distinguishes two main phases, called *generation* and *evaluation*. The *generation* phase, like the *exploration* phase of NMC, generates new combinations of parameter values, while the *evaluation* phase, like the *exploitation* phase of NMC, evaluates the generated combinations. The main difference with NMC is that LSI performs these two phases in sequence instead of interleaving them, and a total predefined budget of available samples $N = N_g + N_e$ is divided among them.

Figure 5 gives the pseudocode for the LSI algorithm. The procedure LsiParametersTuning(N_g, N_e, k) implements the main logic of LSI. The *generation* phase uses up to N_g samples (i.e. MCTS simulations) to generate a set $C^* \subseteq C = V_1 \times ... \times V_n$ of at most k legal combinations of parameters. The

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

procedure Generate($N_g, k$)
Input: Num. samples, $N_g$, for the generation phase.
       Num. 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
        $\boldsymbol{p} \leftarrow$ empty array of size $n$
        $V \leftarrow \bigcup_{i=1}^{n} V_i$
        while $V \neq \emptyset$ do
            $v_i^j \sim \mathcal{D}[\hat{R} \restriction_V]$
            $V \leftarrow V \setminus V_i$
            $\boldsymbol{p}[i] \leftarrow v_i^j$
        $C^* \leftarrow C^* \cup \{\boldsymbol{p}\}$
    return $C^*$
```

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

procedure Evaluate($C^*, N_e$)
Input: Set of parameter combinations to evaluate, $C^*$.
       Num. 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
        $m \leftarrow \left\lfloor \frac{N_e}{|C_i| \lceil \log_2 |C^*| \rceil} \right\rfloor$
        for $m$ times do
            for each $\boldsymbol{p} \in C_i$ do
                $r \leftarrow$ PerformMctsSimulation($\boldsymbol{p}$)
                average expected value of $\boldsymbol{p}$ with $r$
        $C_{i+1} \leftarrow \lceil |C_i|/2 \rceil$ elements with highest estimated value
    return the only combination $\boldsymbol{p} \in C_{\lceil \log_2 |C^*| \rceil}$
```

Fig. 5. Pseudocode for LSI

evaluation phase uses up to N_e samples to evaluate the combinations of values in C^* and recommend the best one, $\boldsymbol{p}^*$. When both phases of LSI are over, the recommended best combination $\boldsymbol{p}^*$ is used to control the rest of the MCTS simulations until the game terminates. The PerformMctsSimulation($\boldsymbol{p}$) procedure, before returning the control to the LSI procedure, takes care of playing a move in the real game if the timeout is reached.

The procedure SideInfo(N_g) constructs the function $\hat{R} : \bigcup_{i=1}^{n} V_i \rightarrow \mathbb{R}$, 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 SideInfo(N_g) divides equally over all the parameter values the total number of generation samples N_g. Every time a parameter value v_i^j is sampled using an MCTS simulation the other parameters are set to random legal 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 $\mathcal{D}[\hat{R} \restriction_V]$, where the first time $V = \bigcup_{i=1}^{n} V_i$ (i.e. all the domain), while 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* [19] 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 considered 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 simulations N that must be set in advance. In GGP is not possible to exactly estimate how many simulations will be performed for a game, thus the game might end before LSI can actually complete its execution or LSI might complete its execution much earlier than the game end. This means that choosing a value for N is not a trivial task. If the value is too high, the search is likely controlled by parameter values selected randomly. 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.

An alternative to deal with this issue is to modify LSI to tune parameters per move instead of per game. In this way the known play clock time can be used to estimate the available budget for the tuning. The search time T available for each move can be divided among a generation phase and an evaluation phase, $T = T_g + Te$, and used to control the execution of the phases of LSI instead of the total simulations budget N. However, preliminary results obtained by testing this strategy showed that it does not improve upon the original LSI implementation.

5 Empirical Evaluation

This section presents an empirical evaluation of the proposed on-line parameter tuning mechanism and a comparison of the discussed allocation strategies. The setup of the performed experiments is presented in Subsect. 5.1, while Subsects. 5.2, 5.3, 5.4 and 5.5 report the obtained results.

5.1 Setup

The on-line parameter tuning mechanism has been implemented in the GGP framework provided by the open source GGP-Base project [32]. The tuning mechanism is used to tune the search parameters of a GGP agent that implements MCTS and uses GRAVE as selection strategy and MAST as play-out strategy.

Below are the MCTS parameters that are tuned on-line, together with their set of possible values:

- The UCT constant $\boldsymbol{C} \in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$.
- The *equivalance parameter* $\boldsymbol{K} \in \{0, 10, 50, 100, 250, 500, 750, 1\,000, 2\,000, \infty\}$ for the GRAVE strategy. Note that this parameter can turn off the GRAVE strategy, because when $K = 0$ the AMAF statistics are not considered and the selection strategy becomes pure UCT.
- The parameter $\boldsymbol{ref} \in \{0, 50, 100, 250, 500, 1\,000, 10\,000, \infty\}$ for the GRAVE strategy. This parameter can control which selection strategy is used. If $ref = 0$ GRAVE behaves exactly like RAVE, while if $ref = \infty$ it behaves like HRAVE [34].
- The parameter $\boldsymbol{\epsilon} \in \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$ for the ϵ-greedy strategy used by MAST. This parameter controls which play-out strategy is used. When $\epsilon = 1$ the strategy becomes completely greedy, while when $\epsilon = 0$ it becomes completely random.

All combinations of values are considered legal, except combinations with $K = 0$, that are legal only if the ref parameter is not considered in the combination. This is because when $K = 0$ the GRAVE strategy is not used, thus the ref parameter has no influence on the search.

The agents implemented[1] for the experiments are the following five:

- $\mathbf{P_{BASE}}$: baseline agent that uses GRAVE and MAST with parameter values $C = 0.2, K = 250, ref = 50, \epsilon = 0.4$). These values were assigned by off-line tuning on the following set of games: *3D Tic Tac Toe*, *Breakthrough*, *Knightthrough*, *Skirmish*, *Battle*, *Chinook*, *Chinese Checkers*.
- $\mathbf{P_{MAB}}$: agent that tunes parameters on-line using the MAB allocation strategy. As selection policy π_{mab}, it uses UCB1 with $C = 0.7$ (a few values were tested and 0.7 proved to perform best). While performing experiments, it was noticed that the allocation strategy of this agent introduces a high overhead on each MCTS simulation because of the exponential number of combinations that it has to check. To alleviate this issue this agent was set to choose a new parameter combination every 10 simulations instead of 1. In this way for each selected combination 10 samples are collected all at once and the overhead is distributed over them.
- $\mathbf{P_{HE}}$: agent that tunes parameters on-line using the HE allocation strategy. When building the HE tree the order of the parameters is randomized before every run of a game. This choice was determined by the fact that experiments with a fixed order for the parameters did not show a particular improvement in the performance of the agent. The policy π_{he} was set to the UCB1 policy with $C = 0.7$.
- $\mathbf{P_{NMC}}$: agent that tunes parameters on-line using the NMC allocation strategy. The policy π_0 is set to be an ϵ-greedy strategy, which performs the exploration phase with probability $\epsilon_0 = 0.75$ and the exploitation phase with probability $1 - \epsilon_0 = 0.25$. These values are the same that are set in [27]. The policies π_l and π_g are both set to UCB1 with $C = 0.7$. In the last series of experiments a variant of this agent is used. This variant sets $\epsilon_0 = 1$ (i.e. it uses only local MABs to choose the parameter values to test) and is identified as $\mathbf{LocalP_{NMC}}$.
- $\mathbf{P_{LSI}}$: agent that tunes parameters on-line using the LSI allocation strategy. The parameters for LSI are set as follows: $N = 100\,000$ (total number of samples, divided among the generation phase, $N_g = 75\,000$, and the evaluation phase, $Ne = 25\,000$), and $k = 2\,000$. The values for N_g and N_e were chosen to keep the proportion between *generation* and *evaluation* samples the same as the proportion between *exploration* and *exploitation* in NMC. The value for N is chosen to be the same for all games and during experiments 100 000 samples showed to be the best choice (lower and higher values decreased the overall performance). However, using 100 000 samples will cause LSI to not be able to finish execution for some of the considered games. A version of LSI that repeats the algorithm for each move in the game using the available search time T as total budget has also been implemented and tested.

[1] Available on request at `https://bitbucket.org/CFSironi/ggp-project`

Table 1. Win percentage of the tuning agents against the off-line tuned agent

Game	$\mathbf{P}_{\mathbf{MAB}}$	$\mathbf{P}_{\mathbf{HE}}$	$\mathbf{P}_{\mathbf{NMC}}$	$\mathbf{P}_{\mathbf{LSI}}$
3DTicTacToe	20.8(±3.38)	34.5(±3.97)	**42.6**(±4.10)	40.7(±4.07)
Breakthrough	8.0(±2.38)	53.6(±4.38)	**57.4**(±4.34)	40.8(±4.31)
Knightthrough	20.0(±3.51)	69.8(±4.03)	**70.6**(±4.00)	53.8(±4.37)
Chinook	21.5(±3.31)	30.8(±3.70)	**34.9**(±3.75)	20.0(±3.27)
ChineseCheckers3	**38.7**(±4.26)	34.9(±4.17)	36.3(±4.20)	36.5(±4.21)
Checkers	8.3(±2.23)	33.6(±3.92)	**37.5**(±3.95)	18.2(±3.17)
Connect 5	13.8(±2.29)	25.9(±3.01)	27.4(±2.91)	**40.7**(±3.23)
Quad	46.8(±4.23)	29.9(±3.76)	35.6(±3.98)	**70.6**(±3.78)
SheepAndWolf	42.6(±4.34)	43.2(±4.35)	45.2(±4.37)	**49.0**(±4.39)
TTCC4 2P	22.4(±3.63)	38.6(±4.15)	**43.5**(±4.21)	24.2(±3.69)
TTCC4 3P	43.9(±4.25)	40.6(±4.14)	41.6(±4.20)	**48.2**(±4.27)
Connect 4	42.0(±4.14)	37.0(±4.09)	43.5(±4.14)	**52.2**(±4.23)
Pentago	26.1(±3.75)	40.6(±4.14)	**45.7**(±4.15)	39.4(±4.09)
Reversi	31.1(±4.00)	**41.5**(±4.26)	38.3(±4.22)	34.3(±4.12)
Avg Win%	27.6(±1.01)	39.6(±1.10)	**42.9**(±1.11)	40.6(±1.11)

Its results are excluded from the paper because it did not show any increase in the performance.

In addition, the last available version of CadiaPlayer[2] [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.

All agents are tested on a set of 14 heterogeneous games: *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*. The GDL descriptions of the games can be downloaded from the GGP-Base repository [31].

For each experiment, two agents at a time are matched against each other. For each game, all possible assignments of agents to the roles of the game 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 been played. Each match runs with $1s$ start clock and $1s$ play clock, except for the experiments that involve CadiaPlayer. In these experiments, CadiaPlayer uses $10s$ start clock and $10s$ play clock while the other agents use $1s$ both for the start clock and the play clock. The reason for this choice is that our agents use a PropNet-based reasoner and can thus perform a higher number of simulations per second than CadiaPlayer.

The results of the experiments always report the average win percentage of one of the two involved agent types with a 95% confidence interval. The average win percentage of an agent type is computed by assigning 1 point for each game where it achieved the highest score, 0 points for each game where the opponent achieved the highest score and 0.5 points for each game where both agents achieved the same score.

[2] Verision of November 18, 2012. Downloaded from the CadiaPlayer project website: `http://cadia.ru.is/wiki/public:cadiaplayer:main`

Table 2. Iterations per second of all agents

Game	$\mathbf{P}_{\mathbf{BASE}}$	$\mathbf{P}_{\mathbf{MAB}}$	$\mathbf{P}_{\mathbf{HE}}$	$\mathbf{P}_{\mathbf{NMC}}$	$\mathbf{P}_{\mathbf{LSI}}$
3DTicTacToe	5259	3782	5918	5864	5945
Breakthrough	4344	3019	4554	4584	4310
Knightthrough	5466	3818	5857	5596	5810
Chinook	3559	2993	3302	3266	3896
ChineseCheckers3	4824	3152	4895	4243	4518
Checkers	630	634	647	651	631
Connect 5	2179	2168	2516	2551	2240
Quad	3614	3118	3890	3837	3934
SheepAndWolf	2381	2069	2423	1952	2239
TTCC4 2P	1401	1433	1527	1532	1406
TTCC4 3P	1970	1525	2108	1936	1910
Connect 4	8218	4488	8494	7915	8289
Pentago	4272	2937	4280	4167	3874
Reversi	287	284	291	287	288

5.2 On-line Tuning Agents vs Off-Line Tuned Agent

This series of experiments evaluated the performance of each of the tuning agents against the baseline agent that is tuned manually off-line. Table 1 shows the obtained results. None of the tuning agents can reach the performance of the off-line tuned agent. $\mathrm{P}_{\mathrm{MAB}}$ shows the worst performance and is always worse than the baseline agent. The poor performance of $\mathrm{P}_{\mathrm{MAB}}$ is due to the fact that the high number of possible values combinations prevents the agent to be able to sample each of them a sufficient number of times to start converging.

Another reason for the poor performance of $\mathrm{P}_{\mathrm{MAB}}$ is that every time a combination must be selected there is quite some computational overhead due to the necessity of iterating over all possible combinations to compute the one with the highest UCB1 value. This reduces the number of simulations that can be performed. Performing the evaluation of each parameter combination using a batch of simulations instead of a single simulation is still not sufficient to increase the performance to the same level as the off-line tuned agent.

Table 2 gives the average median of the number of simulations per second that each of the agents can perform. For most of the games $\mathrm{P}_{\mathrm{MAB}}$ loses a few thousands simulations with respect to the other agents. The other tuning agents, instead, seem to gain a few hundred simulations per second with respect to $\mathrm{P}_{\mathrm{BASE}}$ in most of the games. The explanation for this 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 $\mathrm{P}_{\mathrm{BASE}}$.

P_{HE}, $\mathrm{P}_{\mathrm{NMC}}$ and $\mathrm{P}_{\mathrm{LSI}}$ are close to each other in performance, but $\mathrm{P}_{\mathrm{NMC}}$ seems to be slightly better than the others in most of the games. All three of them show a better performance than $\mathrm{P}_{\mathrm{MAB}}$, and can reach a statistically significant improvement over $\mathrm{P}_{\mathrm{BASE}}$ for one or two games. $\mathrm{P}_{\mathrm{NMC}}$ performed even better than $\mathrm{P}_{\mathrm{BASE}}$ in *Knighttrough* and *Breakthrough*, whereas $\mathrm{P}_{\mathrm{LSI}}$ performed better than $\mathrm{P}_{\mathrm{BASE}}$ in *Quad*.

In most of the games, however, there is a statistically significant worsening of the performance. It could be that the agents keep evaluating sub-optimal combinations and cannot reach the performance level of the off-line tuned agent. Another explanation might be that, by the time they identify optimal combina-

Table 3. Win percentage of $\mathrm{P_{NMC}}$ against all other tuning agents

Game	**vs $\mathrm{P_{MAB}}$**	**vs $\mathrm{P_{HE}}$**	**vs $\mathrm{P_{LSI}}$**
3DTicTacToe	65.3(±4.00)	56.3(±4.19)	47.4(±4.16)
Breakthrough	88.2(±2.83)	62.4(±4.25)	69.0(±4.06)
Knightthrough	84.4(±3.18)	51.4(±4.39)	65.0(±4.19)
Chinook	57.7(±3.82)	56.4(±3.74)	60.2(±3.81)
ChineseCheckers3	44.8(±4.35)	56.2(±4.34)	51.8(±4.37)
Checkers	82.6(±3.06)	59.0(±4.07)	70.6(±3.72)
Connect 5	66.4(±3.39)	50.7(±3.62)	37.2(±3.57)
Quad	39.5(±4.13)	53.9(±4.18)	20.0(±3.35)
SheepAndWolf	49.6(±4.39)	48.0(±4.38)	46.6(±4.38)
TTCC4 2P	71.0(±3.93)	59.6(±4.22)	71.0(±3.93)
TTCC4 3P	47.7(±4.30)	45.9(±4.18)	49.7(±4.31)
Connect 4	37.3(±4.12)	58.1(±4.11)	39.0(±4.10)
Pentago	55.2(±4.17)	52.5(±4.21)	52.7(±4.22)
Reversi	57.6(±4.26)	52.5(±4.28)	56.9(±4.29)
Avg Win%	60.5(±1.10)	54.5(±1.12)	52.6(±1.13)

tions of parameters, $\mathrm{P_{BASE}}$ has already gained an advantage in the game because it was making better decisions from the start due to already tuned parameters.

An additional remark should be made for $\mathrm{P_{LSI}}$. As mentioned in Subsect. 4.2, the performance of LSI depends on the choice for the total number of samples assigned to the algorithm (N). The poor performance of $\mathrm{P_{LSI}}$ in *Checkers* and *Reversi* is due to the fact that these games produce a low number of simulations per second. With a value of 100 000 for the total number of samples, LSI in these game does not even reach the *evaluation* phase and thus selects random parameters for the whole game.

5.3 Evaluation of Best On-Line Tuning Agent

From the previous series of experiments, $\mathrm{P_{NMC}}$ seems to be the agent that achieves the best performance among the on-line tuning agents. As a validation, this series of experiments matches $\mathrm{P_{NMC}}$ against all of them.

Table 3 shows the obtained results. $\mathrm{P_{NMC}}$ is overall better than $\mathrm{P_{MAB}}$, with $\mathrm{P_{MAB}}$ outperforming it only in *Chinese Checkers* and *Connect 4*. Against $\mathrm{P_{HE}}$, it shows to have a better or at least equal performance in all games. $\mathrm{P_{NMC}}$ is also better that $\mathrm{P_{LSI}}$ in most of the games, but for the games for which $\mathrm{P_{LSI}}$ performs better (*Connect 5*, *Quad*, *Connect 4*) the performance gap is consistent.

5.4 Best On-Line Tuning Agent vs CadiaPlayer

In this series of experiments the off-line tuned agent, $\mathrm{P_{BASE}}$ and the best on-line tuning agent, $\mathrm{P_{NMC}}$ are matched against CadiaPlayer. Table 4 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 was encountering some errors while playing them.

The results are in line with the other experiments. The agent $\mathrm{P_{BASE}}$ shows again a better performance than $\mathrm{P_{NMC}}$ against CadiaPlayer for most of the games, except for *Breakthrough* and *Knightthrough*. These are the only two games for which $\mathrm{P_{NMC}}$ proved to be consistently better than $\mathrm{P_{BASE}}$ when the two agents were matched against each other directly.

Table 4. Win percentage of $\mathrm{P}_{\mathrm{BASE}}$ and $\mathrm{P}_{\mathrm{NMC}}$ with 1s start clock and play clock against CadiaPlayer with 10s start clock and play clock

Game	$\mathbf{P}_{\mathbf{BASE}}$	$\mathbf{P}_{\mathbf{NMC}}$
3DTicTacToe	**94.4**(±2.33)	89.6(±2.87)
Breakthrough	60.8(±4.32)	**69.3**(±4.24)
Knightthrough	47.6(±4.78)	**71.0**(±4.45)
Chinook	**80.0**(±3.44)	63.7(±3.85)
Checkers	**90.5**(±2.61)	81.1(±3.36)
Connect 5	**71.4**(±3.27)	48.8(±3.83)
Quad	**98.7**(±1.14)	94.9(±2.11)
SheepAndWolf	**57.7**(±4.35)	50.6(±4.52)
Connect 4	**69.8**(±3.96)	56.5(±4.65)
Pentago	**73.4**(±3.80)	64.7(±4.25)
Avg Win%	**73.4**(±1.25)	68.6(±1.31)

Table 5. Win percentage of $\mathrm{LocalP}_{\mathrm{NMC}}$ and $\mathrm{Best}(\mathrm{P}_{\mathrm{NMC}}, \mathrm{P}_{\mathrm{LSI}})$ against $\mathrm{P}_{\mathrm{BASE}}$

Game	$\mathbf{LocalP}_{\mathbf{NMC}}$	$\mathbf{Best(P_{NMC}, P_{LSI})}$
3DTicTacToe	37.2(±4.03)	**42.6**(±4.10)
Breakthrough	54.8(±4.37)	**57.4**(±4.34)
Knightthrough	70.2(±4.01)	**70.6**(±4.00)
Chinook	30.0(±3.56)	**34.9**(±3.75)
ChineseCheckers3	30.6(±4.03)	**36.5**(±4.21)
Checkers	34.3(±3.95)	**37.5**(±3.95)
Connect 5	30.6(±3.23)	**40.7**(±3.23)
Quad	30.3(±3.79)	**70.6**(±3.78)
SheepAndWolf	45.4(±4.37)	**49.0**(±4.39)
TTCC4 2P	34.8(±4.07)	**43.5**(±4.21)
TTCC4 3P	41.3(±4.09)	**48.2**(±4.27)
Connect 4	36.7(±4.03)	**52.2**(±4.23)
Pentago	37.6(±4.11)	**45.7**(±4.15)
Reversi	33.1(±4.05)	**38.3**(±4.22)
Avg Win%	39.1(±1.09)	**47.7**(±1.12)

Moreover, the difference in performance between $\mathrm{P}_{\mathrm{BASE}}$ and $\mathrm{P}_{\mathrm{NMC}}$ against CadiaPlayer is also comparable to the difference in performance that they showed when matched against each other in the first series of experiments.

5.5 Parameters Inter-dependency

In the last series of experiments the performance of an agent that assumes no inter-dependency between the parameters is compared with the performance that can be obtained by exploiting this inter-dependency.

Table 5 first reports the results of $\mathrm{LocalP}_{\mathrm{NMC}}$. This is the version of $\mathrm{P}_{\mathrm{NMC}}$ that selects parameters combinations using only the local MABs, thus each parameter is always selected independently of the others.

In the last column, the table reports the best performance achieved in previous experiments by either $\mathrm{P}_{\mathrm{NMC}}$ or $\mathrm{P}_{\mathrm{LSI}}$ ($\mathrm{Best}(\mathrm{P}_{\mathrm{NMC}}, \mathrm{P}_{\mathrm{LSI}})$). Both these agents take into account the inter-dependency of the parameters.

These results show how, for each game, at least one allocation strategy (either NMC or LSI) that exploits the inter-dependency of the parameters can outperform a strategy that does not. This can be seen as a confirmation that there is a dependency among the parameters and it should be exploited.

6 Conclusion and Future Work

This paper presented an on-line tuning method for search-control parameters for the MCTS algorithm and evaluated the performance of this method in the context of GGP. Four different sample allocation strategies were introduced and tested for the parameter tuning algorithm, MAB allocation, Hierarchical Expansion, Naïve Monte-Carlo and Linear Side Information.

Results show that, despite having a lower overall performance, the proposed method for on-line parameter tuning can reach a performance almost as good as an off-line tuning approach. This is especially remarkable because only a single run of a game is used to tune the parameters, instead of a few hundred per parameter or per parameter combination. Taking this consideration into account, it may be concluded that the approach proposed in this paper is a valid alternative when there is not sufficient time to tune off-line a high number of parameters with many possible values.

Looking only at the on-line tuning approaches NMC is the one that performs the best. One of the reasons for this is that it tunes the parameters considering their inter-dependency. However, its good performance also depends on the fact that, at the same time, it speeds up the learning process by exploiting the statistics of single parameters to predict the performance of their combinations.

For the same reasons mentioned for NMC, LSI also shows a good performance. However, using it for parameter tuning has a risk: the exact length of a game cannot be known in advance and it is difficult to correctly estimate an appropriate initial budget N. This can negatively influence the performance for games that are too short or for which a small number of simulations per second can be performed.

Future work could look into solving this issue by devising a strategy that chooses an appropriate budget N for each game. For example, a value for N could be selected by estimating the average game length and the average simulations per second that can be performed for the considered game.

Another direction for future work are evolutionary algorithms, which could be used to evolve the set of parameter values over time. In this way the set of available values will not be fixed but will change towards more and more accurate values for the game being considered. Approaches based on evolutionary algorithms have been successfully applied to a problem that is similar to the one of search-control parameter tuning, that is the problem of tuning weights of a state-evaluation function for MCTS [2, 24, 29]. Moreover, in [22] evolutionary algorithms are investigated in the context of automatic game design, where they are used in combination with MABs to evolve the set of parameters that control the characteristics of the generated games.

Acknowledgments. This work is funded by the Netherlands Organisation for Scientific Research (NWO) in the framework of the project GoGeneral, grant number 612.001.121.

References

1. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Machine Learning 47(2-3), 235–256 (2002)
2. Benbassat, A., Sipper, M.: EvoMCTS: A scalable approach for general game learning. IEEE Transactions on Computational Intelligence and AI in Games 6(4), 382–394 (2014)
3. 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)
4. Björnsson, Y., Marsland, T.A.: Learning extension parameters in game-tree search. Information Sciences 154(3), 95–118 (2003)
5. Bouzy, B., Helmstetter, B.: Monte-Carlo Go developments. In: Advances in Computer Games Many Games, Many Challenges. IFIP, vol. 263, pp. 159–174. Kluwer Academic (2003)
6. 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)
7. Brügmann, B.: Monte Carlo Go. Tech. rep., Max Planck Institute of Physics, München, Germany (1993)
8. Burke, E.K., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., Qu, R.: Hyper-heuristics: A survey of the state of the art. Journal of the Operational Research Society 64(12), 1695–1724 (2013)
9. Cazenave, T.: Generalized rapid action value estimation. In: Proceedings of the 24th International Joint Conference on Artificial Intelligence. pp. 754–760. AAAI Press (2015)
10. Chaslot, G.M.J.B., Winands, M.H.M., Szita, I., van den Herik, H.J.: Cross-entropy for Monte-Carlo tree search. ICGA Journal 31(3), 145–156 (2008)
11. Chaslot, G.M.J.B., Winands, M.H.M., van den Herik, H.J., Uiterwijk, J.W.H.M., Bouzy, B.: Progressive strategies for Monte-Carlo tree search. New Mathematics and Natural Computation 4(3), 343–357 (2008)
12. Cole, N., Louis, S.J., Miles, C.: Using a genetic algorithm to tune first-person shooter bots. In: Evolutionary Computation 2004 (CEC2004), Congress on. vol. 1, pp. 139–145. IEEE (2004)
13. Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo tree search. In: van den Herik, 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)
14. Coulom, R.: CLOP: Confident local optimization for noisy black-box parameter tuning. In: Advances in Computer Games. LNCS, vol. 7168, pp. 146–157. Springer (2012)
15. Finnsson, H., Björnsson, Y.: Simulation-based approach to General Game Playing. In: AAAI. vol. 8, pp. 259–264 (2008)
16. Finnsson, H., Björnsson, Y.: Learning simulation control in General Game-Playing agents. In: AAAI. vol. 10, pp. 954–959 (2010)
17. Fürnkranz, J.: Recent advances in machine learning and game playing. ÖGAI Journal 26(2), 19–28 (2007)
18. 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)

19. Karnin, Z., Koren, T., Somekh, O.: Almost optimal exploration in Multi-Armed Bandits. In: Proceedings of the 30th International Conference on Machine Learning. pp. 1238–1246 (2013)
20. Kocsis, L., Szepesvári, C., Winands, M.H.M.: RSPSA: Enhanced parameter optimization in games. In: van den Herik, H.J., Hsu, S.C., Hsu, T.S., Donkers, H. (eds.) Advances in Computer Games Conference (ACG 2005). LNCS, vol. 4250, pp. 39–56. Springer (2006)
21. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: Machine Learning: ECML 2006, LNCS, vol. 4212, pp. 282–293. Springer (2006)
22. 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 2017, Congress on. pp. 2201–2208. IEEE (2017)
23. Levine, J., Congdon, C.B., Ebner, M., Kendall, G., Lucas, S.M., Miikkulainen, R., Schaul, T., Thompson, T.: General video game playing. In: Artificial and Computational Intelligence in Games. Dagstuhl Follow-up, vol. 6, pp. 77–83 (2013)
24. Lucas, S.M., Samothrakis, S., Perez, D.: Fast evolutionary adaptation for Monte Carlo tree search. In: European Conference on the Applications of Evolutionary Computation. pp. 349–360. Springer (2014)
25. Mendes, A., Togelius, J., Nealen, A.: Hyper-heuristic general video game playing. In: Computational Intelligence and Games (CIG), 2016 IEEE Conference on. pp. 94–101. IEEE (2016)
26. Nijssen, J.A.M., Winands, M.H.M.: Enhancements for multi-player Monte-Carlo tree search. In: van den Herik, H.J., Ida, H., Plaat, A. (eds.) Computers and Games, LNCS, vol. 6515, pp. 238–249. Springer (2011)
27. Ontanón, S.: The combinatorial multi-armed bandit problem and its application to real-time strategy games. In: Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment. pp. 58–64. AAAI Press (2013)
28. Ontanón, S.: Combinatorial multi-armed bandits for real-time strategy games. Journal of Artificial Intelligence Research 58, 665–702 (2017)
29. Perez, D., Samothrakis, S., Lucas, S.: Knowledge-based fast evolutionary MCTS for general video game playing. In: Computational Intelligence and Games (CIG), 2014 IEEE Conference on. pp. 68–75. IEEE (2014)
30. Roelofs, G.J.: Action Space Representation in Combinatorial Multi-Armed Bandits. Master's thesis, Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands (2015)
31. Schreiber, S.: Games - base repository. `http://games.ggp.org/base/` (2017)
32. Schreiber, S., Landau, A.: The General Game Playing base package. `https://github.com/ggp-org/ggp-base` (2017)
33. Shleyfman, A., Komenda, A., Domshlak, C.: On combinatorial actions and CMABs with linear side information. In: Proceedings of the Twenty-first European Conference on Artificial Intelligence. pp. 825–830. IOS Press (2014)
34. 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)
35. Świechowski, M., Mańdziuk, J.: Self-adaptation of playing strategies in General Game Playing. IEEE Transactions on Computational Intelligence and AI in Games 6(4), 367–381 (2014)
36. Tak, M.J.W., Winands, M.H.M., Björnsson, Y.: N-grams and the Last-Good-Reply policy applied in General Game Playing. Computational Intelligence and AI in Games, IEEE Transactions on 4(2), 73–83 (2012)