Open access • Journal Article • DOI:10.1109/TCIAIG.2013.2239295

Monte Carlo Search Algorithm Discovery for Single-Player Games — Source link

Francis Maes, David Lupien St-Pierre, Damien Ernst

Institutions: University of Liège

Published on: 11 Jan 2013 - IEEE Transactions on Computational Intelligence and AI in Games (IEEE)

Topics: MCS algorithm, Search algorithm, Monte Carlo tree search, Monte Carlo algorithm and Rejection sampling

Monte Carlo Search Algorithm Discovery for Single-Player Games

Francis Maes, David Lupien St-Pierre, and Damien Ernst

***Abstract*—Much current research in AI and games is being devoted to Monte Carlo search (MCS) algorithms. While the quest for a single unified MCS algorithm that would perform well on all problems is of major interest for AI, practitioners often know in advance the problem they want to solve, and spend plenty of time exploiting this knowledge to customize their MCS algorithm in a problem-driven way. We propose an MCS algorithm discovery scheme to perform this in an automatic and reproducible way. First, we introduce a grammar over MCS algorithms that enables inducing a rich space of candidate algorithms. Afterwards, we search in this space for the algorithm that performs best on average for a given distribution of training problems. We rely on multiarmed bandits to approximately solve this optimization problem. The experiments, generated on three different domains, show that our approach enables discovering algorithms that outperform several well-known MCS algorithms such as upper confidence bounds applied to trees and nested Monte Carlo search. We also show that the discovered algorithms are generally quite robust with respect to changes in the distribution over the training problems.**

***Index Terms*—Algorithm selection, grammar of algorithms, Monte Carlo search (MCS).**

I. Introduction

MONTE CARLO search (MCS) algorithms rely on random simulations to evaluate the quality of states or actions in sequential decision-making problems. Most of the recent progress in MCS algorithms has been obtained by integrating smart procedures to select the simulations to be performed. This has led to, among other things, the upper confidence bounds applied to trees (UCT) algorithm [1] that was popularized thanks to breakthrough results in computer Go [2]. This algorithm relies on a game tree to store simulation statistics and uses this tree to bias the selection of future simulations. While UCT is one way to combine random simulations with tree search techniques, many other approaches are possible. For example, the nested Monte Carlo (NMC) search algorithm [3], which obtained excellent results in the last General Game Playing competition[1] [4], relies on nested levels of search and does not require storing a game tree.

How to best bias the choice of simulations is still an active topic in MCS-related research. Both UCT and NMC are attempts to provide generic techniques that perform well on a wide range of problems and that work with little or no prior knowledge. While working on such generic algorithms is definitely relevant to AI, MCS algorithms are, in practice, widely used in a totally different scenario, in which a significant amount of prior knowledge is available about the game or the sequential decision-making problem to be solved.

People applying MCS techniques typically spend plenty of time exploiting their knowledge of the target problem so as to design more efficient problem-tailored variants of MCS. Among the many ways to do this, one common practice is automatic hyperparameter tuning. By way of example, the parameter $C > 0$ of UCT is in nearly all applications tuned through a more or less automated trial-and-error procedure. While hyperparameter tuning is a simple form of problem-driven algorithm selection, most of the advanced algorithm selection work is done by humans, i.e., by researchers that modify or invent new algorithms to take the specificities of their problem into account.

The comparison and development of new MCS algorithms given a target problem is mostly a manual search process that takes much human time and is error prone. Thanks to modern computing power, automatic discovery is becoming a credible approach for partly automating this process. In order to investigate this research direction, we focus on the simplest case of (fully observable) deterministic single-player games. Our contribution is twofold. First, we introduce a grammar over algorithms that enables generating a rich space of MCS algorithms. It also describes several well-known MCS algorithms, using a particularly compact and elegant description. Second, we propose a methodology based on multiarmed bandits for identifying the best MCS algorithm in this space, for a given distribution over training problems. We test our approach on three different domains. The results show that our method often enables the discovery of new variants of MCS that significantly outperform generic algorithms such as UCT or NMC. We further show the impressive robustness of the discovered algorithms by slightly changing the characteristics of the problem.

This paper is structured as follows. Section II formalizes the class of sequential decision-making problems considered in this paper and formalizes the corresponding MCS algorithm discovery problem. Section III describes our grammar over MCS algorithms and describes several well-known MCS algorithms

Manuscript received August 23, 2012; revised November 16, 2012; accepted January 02, 2013. Date of publication January 11, 2013; date of current version September 11, 2013. This work was supported by the Belgian Network Dynamical Systems, Control, and Optimization (DYSCO), funded by the Interuniversity Attraction Poles Programme, initiated by the Belgian State, Science Policy Office. The scientific responsibility rests with its author(s).

F. Maes was with the Systems and Modeling Research Unit, University of Liège, Liège, Belgium. He is now with the Declarative Languages and Artificial Intelligence Research Group, Catholic University of Leuven (KULeuven), Leuven B-3000, Belgium (e-mail: francis.maes@gmail.com).

D. L. St-Pierre and D. Ernst are with the Department of Electrical Engineering and Computer Science, University of Liège, Liège B-4000, Belgium (e-mail: dlspierre@ulg.ac.be; dernst@ulg.ac.be).

Digital Object Identifier 10.1109/TCIAIG.2013.2239295

[1] http://games.stanford.edu

in terms of this grammar. Section IV formalizes the search for a good MCS algorithm as a multiarmed bandit problem. We experimentally evaluate our approach on different domains in Section V. Finally, we discuss related work in Section VI and conclude in Section VII.

II. Problem Statement

We consider the class of finite-horizon fully observable deterministic sequential decision-making problems. A problem P is a triple (x_1, f, g) where $x_1 \in \mathcal{X}$ is the initial state, f is the transition function, and g is the reward function. The dynamics of a problem are described by

$$x_{t+1} = f(x_t, u_t), \qquad t = 1, 2, \dots, T \tag{1}$$

where, for all t, the state x_t is an element of state space $\mathcal{X}$ and action u_t is an element of the action space. We denote by $\mathcal{U}$ the whole action space and by $\mathcal{U}_x \subset \mathcal{U}$ the subset of actions which are available in state $x \in \mathcal{X}$. In the context of single-player games, x_t denotes the current state of the game and $\mathcal{U}_{x_t}$ are the legal moves in that state. We make no assumptions on the nature of $\mathcal{X}$ but assume that $\mathcal{U}$ is finite. We assume that, when starting from x_1, the system enters a final state after T steps, and we denote by $\mathcal{F} \subset \mathcal{X}$ the set of these final states.[2] Final states $x \in \mathcal{F}$ are associated to rewards $g(x) \in \mathbb{R}$ that should be maximized.

A search algorithm $A(\cdot)$ is a stochastic algorithm that explores the possible sequences of actions to approximately maximize

$$A(P = (x_1, f, g)) \simeq \underset{u_1,\dots,u_T}{\arg\max}\, g(x_{T+1}) \tag{2}$$

subject to $x_{t+1} = f(x_t, u_t)$ and $u_t \in \mathcal{U}_{x_t}$. In order to fulfill this task, the algorithm is given a finite amount of computational time, referred to as the *budget*. To facilitate reproducibility, we focus primarily in this paper on a budget expressed as the maximum number $B > 0$ of sequences $(u_1, \dots, u_T)$ that can be evaluated, or, equivalently, as the number of calls to the reward function $g(\cdot)$. Note, however, that it is trivial in our approach to replace this definition by other budget measures, as illustrated in one of our experiments in which the budget is expressed as an amount of central processing unit (CPU) time.

We express our prior knowledge as a distribution over problems $\mathcal{D}_P$, from which we can sample any number of *training problems* $P \sim \mathcal{D}_P$. The quality of a search algorithm $A^B(\cdot)$ with budget B on this distribution is denoted by $J_A^B(\mathcal{D}_P)$ and is defined as the expected quality of solutions found on problems drawn from $\mathcal{D}_P$

$$J_A^B(\mathcal{D}_P) = \mathsf{E}_{P \sim \mathcal{D}_P}\{\mathsf{E}_{x_{T+1} \sim A^B(P)}\{g(x_{T+1})\}\} \tag{3}$$

where $x_{T+1} \sim A^B(P)$ denotes the final states returned by algorithm A with budget B on problem P.

Given a class of candidate algorithms $\mathcal{A}$ and given budget B, the algorithm discovery problem amounts to selecting an algorithm $A^* \in \mathcal{A}$ of maximal quality

$$A^* = \underset{A \in \mathcal{A}}{\arg\max}\, J_A^B(\mathcal{D}_P). \tag{4}$$

The two main contributions of this paper are: 1) a grammar that enables inducing a rich space $\mathcal{A}$ of candidate MCS algorithms; and 2) an efficient procedure to approximately solve (4).

III. A Grammar for MCS Algorithms

All MCS algorithms share some common underlying general principles: random simulations, lookahead search, time-receding control, and bandit-based selection. The grammar that we introduce in this section aims at capturing these principles in a pure and atomic way. First, we give an overall view of our approach, then present in detail the components of our grammar, and finally describe previously proposed algorithms by using this grammar.

A. Overall View

We call *search components* the elements on which our grammar operates. Formally, a search component is a stochastic algorithm that, when given a partial sequence of actions $(u_1, \dots, u_{t-1})$, generates one or multiple completions $(u_t, \dots, u_T)$ and evaluates them using the reward function $g(\cdot)$. The search components are denoted by $S \in \mathcal{S}$, where $\mathcal{S}$ is the space of all possible search components.

Let S be a particular search component. We define the search algorithm $A_S \in \mathcal{A}$ as the algorithm that, given problem P, executes S repeatedly with an empty partial sequence of actions $()$, until the computational budget is exhausted. Search algorithm A_S then returns the sequence of actions $(u_1, \dots, u_T)$ that led to the highest reward $g(\cdot)$.

In order to generate a rich class of search components—hence a rich class of search algorithms—in an inductive way, we rely on search-component *generators*. Such generators are functions $\Psi : \Theta \rightarrow \mathcal{S}$ that define a search component $S = \Psi(\theta) \in \mathcal{S}$ when given a set of parameters $\theta \in \Theta$. Our grammar is composed of five search-component generators that are defined in Section III-B: $\Psi \in \{\mathrm{simulate}, \mathrm{repeat}, \mathrm{lookahead}, \mathrm{step}, \mathrm{select}\}$. Four of these search-component generators are parametrized by subsearch components. For example, step and $\mathrm{lookahead}$ are functions $\mathcal{S} \rightarrow \mathcal{S}$. These functions can be nested recursively to generate more and more evolved search components. We construct the space of search algorithms $\mathcal{A}$ by performing this in a systematic way, as detailed in Section IV-A.

B. Search Components

Fig. 1 describes our five search-component generators. Note that we distinguish between search-component *inputs* and search-component generator *parameters*. All our search components have the same two inputs: the sequence of already decided actions $(u_1, \dots, u_{t-1})$ and the current state $x_t \in \mathcal{X}$. The parameters differ from one search-component generator to another. For example, $\mathrm{simulate}$ is parametrized by a simulation policy π^{simu} and repeat is parametrized by the number of

[2] In many problems, the time at which the game enters a final state is not fixed, but depends on the actions played so far. It should, however, be noted that it is possible to make these problems fit this fixed finite-time formalism by artificially postponing the end of the game until T. This can be done, for example, by considering that when the game ends before T, a "pseudofinal state" is reached from which, whatever the actions taken, the game will reach the real final state in T.

```
SIMULATE($(u_1, \ldots, u_{t-1}), x_t$)
Param: $\pi^{simu} \in \Pi^{simu}$
for $\tau = t$ to $T$ do
    $u_\tau \sim \pi^{simu}(x_\tau)$
    $x_{\tau+1} \leftarrow f(x_\tau, u_\tau)$
end for
YIELD($(u_1, \ldots, u_T)$)
```

```
REPEAT($(u_1, \ldots, u_{t-1}), x_t$)
Param: $N > 0, S \in \mathcal{S}$
for $i = 1$ to $N$ do
    INVOKE($S, (u_1, \ldots, u_{t-1}), x_t$)
end for
```

```
LOOKAHEAD($(u_1, \ldots, u_{t-1}), x_t$)
Param: $S \in \mathcal{S}$
for $u_t \in \mathcal{U}_{x_t}$ do
    $x_{t+1} \leftarrow f(x_t, u_t)$
    INVOKE($S, (u_1, \ldots, u_t), x_{t+1}$)
end for
```

```
STEP($(u_1, \ldots, u_{t-1}), x_t$)
Param: $S \in \mathcal{S}$
for $\tau = t$ to $T$ do
    INVOKE($S, (u_1, \ldots, u_{\tau-1}), x_\tau$)
    $u_\tau \leftarrow u^*_\tau$
    $x_{\tau+1} \leftarrow f(x_\tau, u_\tau)$
end for
```

```
SELECT($(u_1, \ldots, u_{t-1}), x_t$)
Param: $\pi^{sel} \in \Pi^{sel}, S \in \mathcal{S}$
for $\tau = t$ to $T$ do                                        ▷ Select
    $u_\tau \sim \pi^{sel}(x)$
    $x_{\tau+1} \leftarrow f(x_\tau, u_\tau)$
    if $n(x_{\tau+1}) = 0$ then
        break
    end if
end for
$t_{leaf} \leftarrow \tau$
INVOKE($S, (u_1, \ldots, u_{t_{leaf}}), x_{t_{leaf}+1}$)                 ▷ Sub-search
for $\tau = t_{leaf}$ to 1 do                                   ▷ Backpropagate
    $n(x_{\tau+1}) \leftarrow n(x_{\tau+1}) + 1$
    $n(x_\tau, u_\tau) \leftarrow n(x_\tau, u_\tau) + 1$
    $s(x_\tau, u_\tau) \leftarrow s(x_\tau, u_\tau) + r^*$
end for
$n(x_1) \leftarrow n(x_1) + 1$
```

Fig. 1. Search-component generators.

repetitions $N > 0$ and by a subsearch component. We now give a detailed description of these search-component generators.

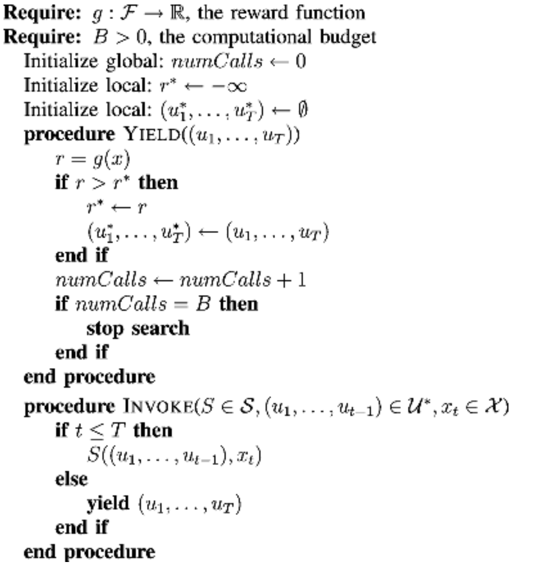
1) Simulate: The simulate generator is parametrized by a policy $\pi^{\text{simu}} \in \Pi^{\text{simu}}$, which is a stochastic mapping from states to actions: $u \sim \pi^{\text{simu}}(x)$. In order to generate the completion $(u_t, \ldots, u_T)$, $\text{simulate}(\pi^{\text{simu}})$ repeatedly samples actions u_τ according to $\pi^{\text{simu}}(x_\tau)$ and performs transitions $x_{\tau+1} = f(x_\tau, u_\tau)$ until reaching a final state. A default choice for the simulation policy is the uniformly random policy, defined as

$$\mathsf{E}\{\pi^{\text{random}}(x) = u\} = \begin{cases} \frac{1}{|\mathcal{U}_x|}, & \text{if } u \in \mathcal{U}_x \\ 0, & \text{otherwise.} \end{cases} \quad (5)$$

Once the completion $(u_t, \ldots, u_T)$ is fulfilled, the whole sequence $(u_1, \ldots, u_T)$ is *yielded*. This operation is detailed in Fig. 2 and proceeds as follows: 1) it computes the reward of the final state x_{T+1}; 2) if the reward is larger than the largest reward found previously, it replaces the best current solution; and 3) if budget B is exhausted, it stops the search.

Since algorithm A_P repeats P until the budget is exhausted, the search algorithm $A_{\text{simulate}(\pi^{\text{simu}})} \in \mathcal{A}$ is the algorithm that samples B random trajectories $(u_1, \ldots, u_T)$, evaluates each of the final state rewards $g(x_{T+1})$, and returns the best found final state. This simple random search algorithm is sometimes called *iterative sampling* [5].

Note that, in the yield procedure, the variables relative to the best current solution [r^* and $(u^*_1, \ldots, u^*_T)$] are defined locally for each search component, whereas the numCalls counter is global to the search algorithm. This means that if S is a search



```
Require: $g : \mathcal{F} \to \mathbb{R}$, the reward function
Require: $B > 0$, the computational budget
  Initialize global: $numCalls \leftarrow 0$
  Initialize local: $r^* \leftarrow -\infty$
  Initialize local: $(u_1^*, \ldots, u_T^*) \leftarrow \emptyset$
  procedure YIELD($(u_1, \ldots, u_T)$)
      $r = g(x)$
      if $r > r^*$ then
          $r^* \leftarrow r$
          $(u_1^*, \ldots, u_T^*) \leftarrow (u_1, \ldots, u_T)$
      end if
      $numCalls \leftarrow numCalls + 1$
      if $numCalls = B$ then
          stop search
      end if
  end procedure
  procedure INVOKE($S \in \mathcal{S}, (u_1, \ldots, u_{t-1}) \in \mathcal{U}^*, x_t \in \mathcal{X}$)
      if $t \leq T$ then
          $S((u_1, \ldots, u_{t-1}), x_t)$
      else
          yield $(u_1, \ldots, u_T)$
      end if
  end procedure
```

Fig. 2. Yield and invoke commands.

component composed of different nested levels of search (see the examples below), the best current solution is kept in memory at each level of the search.

2) Repeat: Given a positive integer $N > 0$ and a search component $S \in \mathcal{S}$, $\text{repeat}(N, S)$ is the search component that repeats N times the search component S. For example, $S = \text{repeat}(10, \text{simulate}(\pi^{\text{simu}}))$ is the search component that draws ten random simulations using π^{simu}. The corresponding search algorithm A_S is again iterative sampling, since search algorithms repeat their search component until the budget is exhausted. In Fig. 1, we use the invoke operation each time a search component calls a subsearch component. This operation is detailed in Fig. 2 and ensures that no subsearch algorithm is called when a final state is reached, i.e., when $t = T + 1$.

3) Lookahead: For each legal move $u_t \in \mathcal{U}_{x_t}$, $\text{lookahead}(S)$ computes the successor state $x_{t+1} = f(x_t, u_t)$ and runs the subsearch component $S \in \mathcal{S}$ starting from the sequence $(u_1, \ldots, u_t)$. For example, $\text{lookahead}(\text{simulate}(\pi^{\text{simu}}))$ is the search component that, given the partial sequence $(u_1, \ldots, u_{t-1})$, generates one random trajectory for each legal next action $u_t \in \mathcal{U}_{x_t}$. Multiple-step lookahead search strategies naturally write themselves with nested calls to lookahead. As an example, $\text{lookahead}(\text{lookahead}(\text{lookahead}(\text{simulate}(\pi^{\text{simu}}))))$ is a search component that runs one random trajectory per legal combination of the three next actions (u_t, u_{t+1}, u_{t+2}).

4) Step: For each remaining time step $\tau \in [t, T]$, $\text{step}(S)$ runs the subsearch component S, extracts the action u_τ from $(u_1^*, \ldots, u_T^*)$ (the best currently found action sequence; see Fig. 2), and performs transition $x_{\tau+1} = f(x_\tau, u_\tau)$. The search-component generator step enables implementing time receding search mechanisms, e.g., $\text{step}(\text{repeat}(100, \text{simulate}(\pi^{\text{simu}})))$ is the search component that selects the actions $(u_1, \ldots, u_T)$ one by one, using 100 random trajectories to select each action. As a more evolved example, $\text{step}(\text{lookahead}(\text{lookahead}(\text{repeat}(10, \text{simulation}(\pi^{\text{simu}})))))$ is a time receding strategy that performs ten random simulations for each two first actions (u_t, u_{t+1}) to decide which action u_t to select.

5) Select: This search-component generator implements most of the behavior of a Monte Carlo tree search (MCTS, [1]). It relies on a game tree, which is a nonuniform lookahead tree with nodes corresponding to states and edges corresponding to transitions. The role of this tree is twofold: it stores statistics on the outcomes of subsearches and it is used to bias subsearches toward promising sequences of actions. A search component $\text{select}(\pi^{\text{sel}}, S)$ proceeds in three steps: the *selection* step relies on the statistics stored in the game tree to select a (typically small) subsequence of actions $(u_t, \ldots, u_{t_{\text{leaf}}})$, the *subsearch* step invokes the subsearch component $S \in \mathcal{S}$ starting from $(u_1, \ldots, u_{t_{\text{leaf}}})$, and the *backpropagation* step updates the statistics to take into account the subsearch result.

We use the following notation to denote the information stored by the lookahead tree: $n(x, u)$ is the number of times the action u was selected in state x, $s(x, u)$ is the sum of rewards that were obtained when running *subsearch* after having selected action u in state x, and $n(x)$ is the number of times state x was selected: $n(x) = \sum_{u \in \mathcal{U}_x} n(x, u)$. In order to quantify the quality of a subsearch, we rely on the reward of the best solution that was tried during that subsearch: $r^* = \max g(x)$. In the simplest case, when the subsearch component is $S = \text{simulate}(\pi^{\text{simu}})$, r^* is the reward associated to the final state obtained by making the random simulation with policy π^{simu}, as usual in MCTS. In order to select the first actions, *selection* relies on a *selection policy* $\pi^{\text{sel}} \in \Pi^{\text{sel}}$, which is a stochastic function that, when given all stored information related to state x [i.e., $n(x)$, $n(x, u)$, and $s(x, u)$, $\forall\, u \in \mathcal{U}_x$], selects an action $u \in \mathcal{U}_x$. The selection policy has two contradictory goals to pursue: *exploration*, trying new sequences of actions to increase knowledge, and *exploitation*, using current knowledge to bias computational efforts toward promising sequences of actions. Such exploration/exploitation dilemmas are usually formalized as a multiarmed bandit problem, hence π^{sel} is typically one of policies commonly found in the multiarmed bandit literature. Possibly the most well-known such policy is UCB-1 [6]

$$\pi_C^{\text{ucb}-1}(x) = \underset{u \in \mathcal{U}_x}{\arg\max} \frac{s(x, u)}{n(x, u)} + C \sqrt{\frac{\ln n(x)}{n(x, u)}} \tag{6}$$

where division by zero returns $+\infty$ and where $C > 0$ is a hyperparameter that enables the control of the exploration/exploitation tradeoff.

C. Description of Previously Proposed Algorithms

Our grammar enables generating a large class of MCS algorithms, which includes several already proposed algorithms. We now overview these algorithms, which can be described particularly compactly and elegantly thanks to our grammar.

- The simplest Monte Carlo algorithm in our class is *iterative sampling*. This algorithm draws random simulations until the computational time is elapsed and returns the best solution found

$$\text{is} = \text{simulate}(\pi^{\text{simu}}). \tag{7}$$

- In general, iterative sampling is used during a certain period to decide which action to select (or which move to play) at each step of the decision problem. The corresponding search component is

$$\text{is}' = \text{step}(\text{repeat}(N, \text{simulate}(\pi^{\text{simu}}))) \tag{8}$$

  where N is the number of simulations performed for each decision step.
- The *reflexive Monte Carlo* search algorithm introduced in [7] proposes using an MCS of a given level to improve the search of the upper level. The proposed algorithm can be described as follows:

$$\begin{aligned}&\text{rmc}(N_1, N_2)\\&= \text{step}(\text{repeat}(N_1, \text{step}(\text{repeat}(N_2, \text{simulate}(\pi^{simu})))))\end{aligned} \tag{9}$$

  where N_1 and N_2 are called the number of meta-games and the number of games, respectively.
- The nested Monte Carlo (NMC) search algorithm [3] is a recursively defined algorithm generalizing the ideas of reflexive Monte Carlo search. NMC can be described in a very natural way by our grammar. The basic search level $l = 0$ of NMC simply performs a random simulation

$$\text{nmc}(0) = \text{simulate}(\pi^{\text{random}}). \tag{10}$$

  Level $l > 0$ of NMC relies on level $l - 1$ in the following way:

$$\text{nmc}(l) = \text{step}(\text{lookahead}(\text{nmc}(l-1))). \tag{11}$$

- Single-player MCTS [8]–[10] selects actions one after the other. In order to select one action, it relies on select combined with random simulations. The corresponding search component is thus

$$\begin{aligned}&\text{mcts}(\pi^{\text{sel}}, \pi^{\text{simu}}, N)\\&= \text{step}(\text{repeat}(N, \text{select}(\pi^{\text{sel}}, \text{simulate}(\pi^{\text{simu}}))))\end{aligned} \tag{12}$$

  where N is the number of iterations allocated to each decision step. UCT is one of the best known variants of MCTS. It relies on the $\pi_C^{\text{ucb}-1}$ selection policy and is generally used with a uniformly random simulation policy

$$\text{uct}(C, N) = \text{mcts}(\pi_C^{\text{ucb}-1}, \pi^{\text{random}}, N). \tag{13}$$

- In the spirit of the work on NMC, Chaslot *et al.* [11] proposed the meta MCTS approach, which replaces the simulation part of an upper level MCTS algorithm by a whole lower level MCTS algorithm. While they presented this approach in the context of two-player games, we can describe its equivalent for single-player games with our grammar

$$\begin{aligned}&\text{metamcts}(\pi^{\text{sel}}, \pi^{\text{simu}}, N_1, N_2)\\&\quad= \text{step}(\text{repeat}(N_1, \text{select}(\pi^{\text{sel}}, \text{mcts}(\pi^{\text{sel}}, \pi^{\text{simu}}, N_2))\end{aligned} \tag{14}$$

  where N_1 and N_2 are the budgets for the higher level and lower level MCTS algorithms, respectively.

In addition to offering a framework for describing these already proposed algorithms, our grammar enables generating a large number of new hybrid MCS variants. We give, in Section IV, a procedure to automatically identify the best such variant for a given problem.

IV. Bandit-Based Algorithm Discovery

We now move to the problem of solving (4), i.e., of finding, for a given problem, the best algorithm A from among a large class $\mathcal{A}$ of algorithms derived with the grammar previously defined. Solving this algorithm discovery problem exactly is impossible in the general case since the objective function involves two infinite expectations: one over problems $P \sim \mathcal{D}_P$ and another over the outcomes of the algorithm. In order to approximately solve (4), we adopt the formalism of multiarmed bandits and proceed in two steps: first, we construct a finite set of candidate algorithms $\mathcal{A}_{D,\Gamma} \subset \mathcal{A}$ (Section IV-A), and then treat each of these algorithms as an arm and use a multiarmed bandit policy to select how to allocate computational time to the performance estimation of the different algorithms (Section IV-B). It is worth mentioning that this two-step approach follows a general methodology for automatic discovery that we already successfully applied to multiarmed bandit policy discovery [12], [13], reinforcement learning policy discovery [14], and optimal control policy discovery [15].

A. Construction of the Algorithm Space

We measure the complexity of a search component $S \in \mathcal{S}$ using its *depth*, defined as the number of nested search components constituting S, and denote this quantity by $\text{depth}(S)$. For example, $\text{depth}(\text{simulate}(\pi^{\text{simu}}))$ is 1, $\text{depth}(\text{uct})$ is 4, and $\text{depth}(\text{nmc}(3))$ is 7.

Note that simulate, repeat, and select have parameters which are not search components: the simulation policy π^{simu}, the number of repetitions N, and the selection policy π^{sel}, respectively. In order to generate a finite set of algorithms using our grammar, we rely on predefined finite sets of possible values for each of these parameters. We denote by Γ the set of these finite domains. The discrete set $\mathcal{A}_{D,\Gamma}$ is constructed by enumerating all possible algorithms up to depth D with constants Γ, and is pruned using the following rules.

- *Canonization of repeat:* Both search components $S_1 = \text{step}(\text{repeat}(2, \text{repeat}(5, S_{\text{sub}})))$ and $S_2 = \text{step}(\text{repeat}(5, \text{repeat}(2, S_{\text{sub}})))$ involve running S_{sub} ten times at each step. In order to avoid having this kind of algorithm duplicated, we collapse nested repeat components into single repeat components. With this rule, S_1 and S_2 both reduce to $\text{step}(\text{repeat}(10, S_{\text{sub}}))$.
- *Removal of nested selects:* A search component such as $\text{select}(\pi^{\text{sel}}, \text{select}(\pi^{\text{sel}}, S))$ is ill-defined, since the inner select will be called with a different initial state x_t each time, making it behave randomly. Therefore, we exclude search components involving two directly nested select's.
- *Removal of repeat-as-root:* Remember that the MCS algorithm $A_S \in \mathcal{A}$ runs S repeatedly until the computational

TABLE I
UNIQUE ALGORITHMS UP TO DEPTH 3

Depth 1–2	Depth 3	
sim	lookahead(repeat(2, sim))	step(repeat(2, sim))
	lookahead(repeat(10, sim))	step(repeat(10, sim))
lookahead(sim)	lookahead(lookahead(sim))	step(lookahead(sim))
step(sim)	lookahead(step(sim))	step(step(sim))
select(sim)	lookahead(select(sim))	step(select(sim))
	select(repeat(2, sim))	select(repeat(10, sim))
	select(lookahead(sim))	select(step(sim))

budget is exhausted. Due to this repetition, algorithms such as $A_{\mathrm{simulate}(\pi^{\mathrm{simu}})}$ and $A_{\mathrm{repeat}(10,\mathrm{simulate}(\pi^{\mathrm{simu}}))}$ are equivalent. To remove these duplicates, we reject all search components whose "root" is repeat.

In the following, ν denotes the cardinality of the set of candidate algorithms: $\mathcal{A}_{D,\Gamma} = \{A_1, \dots, A_\nu\}$. To illustrate the construction of this set, consider a simple case where the maximum depth is $D = 3$ and where constants Γ are $\pi^{\mathrm{simu}} = \pi^{\mathrm{random}}$, $N \in \{2, 10\}$, and $\pi^{\mathrm{sel}} = \pi_C^{\mathrm{ucb-1}}$. The corresponding space $\mathcal{A}_{D,\Gamma}$ contains $\nu = 18$ algorithms. These algorithms are given in Table I, where we use sim as an abbreviation for $\mathrm{simulate}(\pi^{\mathrm{simu}})$.

B. Bandit-Based Algorithm Discovery

One simple approach to approximately solve (4) is to estimate the objective function through an empirical mean computed using a finite set of training problems $\{P^{(1)}, \dots, P^{(M)}\}$, drawn from $\mathcal{D}_P$

$$J_A^B(\mathcal{D}_P) \simeq \frac{1}{M} \sum_{i=1}^{M} g(x_{T+1}) | x_{T+1} \sim A^B(P^{(i)}) \qquad (15)$$

where x_{T+1} denotes one outcome of algorithm A with budget B on problem $P^{(i)}$. To solve (4), one can then compute this approximated objective function for all algorithms $A \in \mathcal{A}_{D,\Gamma}$ and simply return the algorithm with the highest score. While extremely simple to implement, such an approach often requires an excessively large number of samples M to work well, since the variance of $g(\cdot)$ may be quite large.

In order to optimize (4) in a smarter way, we propose to formalize this problem as a multiarmed bandit problem. To each algorithm $A_k \in \mathcal{A}_{D,\Gamma}$, we associate an arm. Pulling the arm k for the t_kth time involves selecting problem $P^{(t_k)}$ and running algorithm A_k once on this problem. This leads to a reward associated to arm k whose value is reward $g(x_{T+1})$ that comes with solution x_{T+1} found by algorithm A_k. The purpose of multiarmed bandit algorithms is to process the sequence of observed rewards to select in a smart way the next algorithm to be tried, so that when the time allocated to algorithm discovery is exhausted, one (or several) high-quality algorithm(s) can be identified. How to select arms so as to identify the best one in a finite amount of time is known as the *pure exploration* multiarmed bandit problem [16]. It has been shown that index-based policies based on upper confidence bounds such as UCB-1 were also good policies for solving pure exploration bandit problems. Our optimization procedure works thus by repeatedly playing arms according to such a policy. In our experiments, we perform a fixed number of such iterations. In practice, this multiarmed bandit approach can provide an answer at anytime, returning algorithm A_k with the currently highest empirical reward mean.

C. Discussion

Note that other approaches could be considered for solving our algorithm discovery problem. In particular, optimization over expression spaces induced by a grammar such as ours is often solved using genetic programming (GP) [17]. GP works by evolving a population of solutions, which, in our case, would be MCS algorithms. At each iteration, the current population is evaluated, the least worthwhile solutions are removed, and the best solutions are used to construct new candidates using mutation and crossover operations. Most existing GP algorithms assume that the objective function is (at least approximately) deterministic. One major advantage of the bandit-based approach is to natively take into account the stochasticity of the objective function and its decomposability into problems. Thanks to the bandit formulation, badly performing algorithms are quickly rejected and the computational power is more and more focused on the most promising algorithms.

The main strengths of our bandit-based approach are the following. First, it is simple to implement and does not require entering into the details of complex mutation and crossover operators. Second, it has only one hyperparameter (the exploration/exploitation coefficient). Finally, since it is based on exhaustive search and on multiarmed bandit theory, formal guarantees can easily be derived to bound the regret, i.e., the difference between the performance of the best algorithm and the performance of the algorithm discovered [16], [18], [19].

Our approach is restricted to relatively small depths D since it relies on exhaustive search. In our case, we believe that many interesting MCS algorithms can be described using search components with low depth. In our experiments, we used $D = 5$, which already provides many original hybrid algorithms that deserve further research. Note that GP algorithms do not suffer from such a limit, since they are able to generate deep and complex solutions through mutation and crossover of smaller solutions. If limit $D = 5$ proved too restrictive, a major way of improvement would thus consist in combining the idea of bandits with those of GP. In this spirit, Hoock *et al.* [20] recently proposed a hybrid approach in which the selection of the members of a new population is posed as a multiarmed bandit problem. This enables combining the best of the two approaches: multiarmed bandits enable natively taking into account the stochasticity and decomposability of the objective function, while GP crossover and mutation operators are used to generate new candidates dynamically in a smart way.

V. EXPERIMENTS

We now apply our automatic algorithm discovery approach to three different testbeds: *Sudoku*, *Symbolic Regression*, and *Morpion Solitaire*. The aim of our experiments was to show that our approach discovers MCS algorithms that outperform several generic (problem-independent) MCS algorithms: outperform them on the training instances, on new testing instances, and even on instances drawn from distributions different from the original distribution used for the learning.

First, we describe the experimental protocol in Section V-A. We perform a detailed study of the behavior of our approach applied to the *Sudoku* domain in Section V-B. Sections V-C and V-D then give the results obtained on the other two domains. Finally, Section V-E gives an overall discussion of our results.

A. Protocol

We now describe the experimental protocol that will be used in the remainder of this section.

1) Generic Algorithms: The generic algorithms are NMC, UCT, lookahead search, and iterative sampling. The search components for NMC (nmc), UCT (uct), and iterative sampling (is) have already been defined in Section III-C. The search component for lookahead search of level $l > 0$ is defined by $\mathrm{la}(l) = \mathrm{step}(\mathrm{larec}(l))$, where

$$\mathrm{larec}(l) = \begin{cases} \mathrm{lookahead}(\mathrm{larec}(l-1)), & \text{if } l > 0 \\ \mathrm{simulate}(\pi^{\mathrm{random}}), & \text{otherwise.} \end{cases} \quad (16)$$

For both $\mathrm{la}(\cdot)$ and $\mathrm{nmc}(\cdot)$, we try all values within the range $[1, 5]$ for the level parameter. Note that $\mathrm{la}(1)$ and $\mathrm{nmc}(1)$ are equivalent, since both are defined by the search component $\mathrm{step}(\mathrm{lookahead}(\mathrm{simulate}(\pi^{\mathrm{random}})))$. For $\mathrm{uct}(\cdot)$, we try the following values of C: $\{0, 0.3, 0.5, 1.0\}$ and set the budget per step to B/T, where B is the total budget and T is the horizon of the problem. This leads to the following set of generic algorithms: $\{\mathrm{nmc}(2)$, $\mathrm{nmc}(3)$, $\mathrm{nmc}(4)$, $\mathrm{nmc}(5)$, is, $\mathrm{la}(1)$, $\mathrm{la}(2)$, $\mathrm{la}(3)$, $\mathrm{la}(4)$, $\mathrm{la}(5)$, $\mathrm{uct}(0)$, $\mathrm{uct}(0.3)$, $\mathrm{uct}(0.5)$, and $\mathrm{uct}(1)\}$. Note that we omit the (B/T) parameter in uct for the sake of conciseness.

2) Discovered Algorithms: In order to generate the set of candidate algorithms, we used the following constants Γ: repeat can be used with 2, 5, 10, or 100 repetitions; and select relies on the UCB1 selection policy from (6) with constants $\{0, 0.3, 0.5, 1.0\}$. We create a pool of algorithms by exhaustively generating all possible combinations of the search components up to depth $D = 5$. We apply the pruning rules described in Section IV-A, which results in a set of $\nu = 3155$ candidate MCS algorithms.

Algorithm Discovery: In order to carry out the algorithm discovery, we used a UCB policy for $100 \times \nu$ time steps, i.e., each candidate algorithm was executed 100 times on average. As discussed in Section IV-B, each bandit step involves running one of the candidate algorithms on a problem $P \sim \mathcal{D}_P$. We refer to $\mathcal{D}_P$ as the *training distribution* in the following. Once we have played the UCB policy for $100 \times \nu$ time steps, we sort the algorithms by their average training performance and report the ten best algorithms.

3) Evaluation: Since algorithm discovery is a form of "learning from example," care must be taken against overfitting issues. Indeed, the discovered algorithms may perform well on the training problems P while performing poorly on other problems drawn from $\mathcal{D}_P$. Therefore, to evaluate the MCS algorithms, we used a set of 10 000 *testing problems* $P \sim \mathcal{D}_P$, which are different from the training problems. We then evaluate the score of an algorithm as the mean performance obtained when running it once on each testing problem.

In each domain, we further test the algorithms either by changing the budget B and/or by using a new distribution $\mathcal{D}'_P$ that differs from the training distribution $\mathcal{D}_P$. In each such experiment, we draw 10 000 problems from $\mathcal{D}'_P$ and run the algorithm once on each problem.

In one domain (*Morpion Solitaire*), we used a particular case of our general setting, in which there was a single training problem P, i.e., the distribution $\mathcal{D}_P$ was degenerate and always returned the same P. In this case, we focused our analysis on the robustness of the discovered algorithms when tested on a new problem P' and/or with a new budget B.

4) Presentation of the Results: For each domain, we present the results in a table in which the algorithms have been sorted according to their *testing* scores on $\mathcal{D}_P$. In each column of these tables, we underline both the best generic algorithm and the best discovered algorithm and show in bold all cases in which a discovered algorithm outperforms all tested generic algorithms. Furthermore, we performed an unpaired t-test between each discovered algorithm and the best generic algorithm. We display significant results (p-value lower than 0.05) by circumscribing them with stars. As in Table I, we use sim as an abbreviation for $\mathrm{simulate}(\pi^{\mathrm{simu}})$ in this section.

B. Sudoku

Sudoku, a Japanese term meaning "singular number," is a popular puzzle played around the world. The *Sudoku* puzzle is made of a grid of $G^2 \times G^2$ cells, which is structured into blocks of size $G \times G$. When starting the puzzle, some cells are already filled in and the objective is to fill in the remaining cells with the numbers 1 through G^2 so that:

- no row contains two instances of the same number;
- no column contains two instances of the same number;
- no block contains two instances of the same number.

Sudoku is of particular interest in our case because each *Sudoku* grid corresponds to a different initial state x_1. Thus, a good algorithm $A(\cdot)$ is one that intrinsically has the versatility to face a wide variety of *Sudoku* grids.

In our implementation, we maintain for each cell the list of numbers that could be put in that cell without violating any of the three previous rules. If one of these lists becomes empty, then the grid cannot be solved and we pass to a final state (see footnote 2). Otherwise, we select the subset of cells whose number list has the lowest cardinality, and define one action $u \in \mathcal{U}_x$ per possible number in each of these cells (as in [3]). The reward associated to a final state is its proportion of filled cells, hence a reward of 1 is associated to a perfectly filled grid.

Algorithm Discovery: We sample the initial states x_1 by filling 33% randomly selected cells as proposed in [3]. We denote by *Sudoku(G)* the distribution over *Sudoku* problems obtained with this procedure (in the case of $G^2 \times G^2$ games). Even though *Sudoku* is most usually played with $G = 3$ [21], we carry out the algorithm discovery with $G = 4$ to make the problem more difficult. Our training distribution was thus $\mathcal{D}_P =$ *Sudoku(4)* and we used a training budget of $B = 1000$ evaluations. To evaluate the performance and robustness of the algorithms found, we tested the MCS algorithms on two distributions: $\mathcal{D}_P =$ *Sudoku(4)* and $\mathcal{D}'_P =$ *Sudoku(5)*, using a budget of $B = 1000$.

Table II presents the results, where the scores are the average number of filled cells, which is given by the reward times

TABLE II
RANKING AND ROBUSTNESS OF ALGORITHMS DISCOVERED WHEN APPLIED TO *SUDOKU*

Name	Search Component	Rank	Sudoku(4)	Sudoku(5)
Dis#8	step(select(repeat(select(sim, 0.5), 5), 0))	1	**198.9**	487.2
Dis#2	step(repeat(step(repeat(sim, 5)), 10))	2	**198.8**	486.2
Dis#6	step(**step(repeat(select(sim, 0), 5))**)	2	**198.8**	486.2
uct(0)		4	198.7	494.4
uct(0.3)		4	198.7	493.3
Dis#7	lookahead(**step(repeat(select(sim, 0.3), 5))**)	6	198.6	486.4
uct(0.5)		6	198.6	492.7
Dis#1	select(**step(repeat(select(sim, 1), 5))**, 1)	6	198.6	485.7
Dis#10	select(**step(repeat(select(sim, 0.3), 5))**)	9	198.5	485.9
Dis#3	step(select(step(sim), 1))	10	198.4	493.7
Dis#4	step(step(step(select(sim, 0.5))))	11	198.3	493.4
Dis#5	select(step(repeat(sim, 5)), 0.5)	11	198.3	486.3
Dis#9	lookahead(step(step(select(sim, 1))))	13	198.1	492.8
uct(1)		13	198.1	486.9
nmc(3)		15	196.7	429.7
la(1)		16	195.6	430.1
nmc(4)		17	195.4	430.4
nmc(2)		18	195.3	430.3
nmc(5)		19	191.3	426.8
la(2)		20	174.4	391.1
la(4)		21	169.2	388.5
is		22	169.1	388.5
la(5)		23	168.3	386.9
la(3)		24	167.1	389.1

the total number of cells G^4. The best generic algorithms on *Sudoku(4)* are $\mathrm{uct}(0)$ and $\mathrm{uct}(0.3)$, with an average score of 198.7. We discover three algorithms that have a better average score (198.8 and 198.9) than $\mathrm{uct}(0)$, but, due to a very large variance on this problem (some *Sudoku* grids are far more easy than others), we could not show this difference to be significant. Although the discovered algorithms are not significantly better than $\mathrm{uct}(0)$, none of them are significantly worse than this baseline. Furthermore, all ten discovered algorithms are significantly better than all the other non-uct baselines. Interestingly, four out of the ten discovered algorithms rely on the uct pattern—$\mathrm{step}(\mathrm{repeat}(\mathrm{select}(\mathrm{sim}, \cdot), \cdot))$—as shown in bold in the table.

When running the algorithms on the *Sudoku(5)* games, the best algorithm is still $\mathrm{uct}(0)$, with an average score of 494.4. This score is slightly above the score of the best discovered algorithm (493.7). However, all ten discovered algorithms are still significantly better than the non-uct generic algorithms. This shows that good algorithms with *Sudoku(4)* are still reasonably good for *Sudoku(5)*.

1) Repeatability: In order to evaluate the stability of the results produced by the bandit algorithm, we performed five runs of algorithm discovery with different random seeds and compared the resulting top-tens. What we observe is that our space contains a huge number of MCS algorithms performing nearly equivalently on our distribution of *Sudoku* problems. In consequence, different runs of the discovery algorithm produce different subsets of these nearly equivalent algorithms. Since we observed that small changes in the constants of repeat and select often have a negligible effect, we grouped the discovered algorithms by structure, i.e., by ignoring the precise values of their constants. Table III reports the number of occurrences of each search-component structure among the five top-tens. We observe that uct was discovered in five cases out of 50 and that the uct pattern is part of 24 discovered algorithms.

2) Time-Based Budget: Since we expressed the budget as the number of calls to the reward function $g(\cdot)$, algorithms that take more time to select their actions may be favored. To evaluate the extent of this potential bias, we performed an experiment by setting the budget to a fixed amount of CPU time. With our C++ implementation, on a 1.9-Ghz computer, about ≈ 350 *Sudoku(4)* random simulations can be performed per second. In order to have comparable results with those obtained previously, we thus set our budget to $B = (1000/350) \approx 2.8$ s, during both algorithm discovery and evaluation.

TABLE III
REPEATABILITY ANALYSIS

Search Component Structure	Occurrences in the top-ten
select(**step(repeat(select(sim)))**)	11
step(**step(repeat(select(sim)))**)	6
step(select(repeat(select(sim))))	5
step(repeat(select(sim)))	5
select(step(repeat(sim)))	2
select(step(select(repeat(sim))))	2
step(select(step(select(sim))))	2
step(step(select(repeat(sim))))	2
step(repeat(step(repeat(sim))))	2
lookahead(**step(repeat(select(sim)))**)	2
step(repeat(step(repeat(sim))))	2
select(repeat(step(repeat(sim))))	1
select(step(repeat(sim)))	1
lookahead(step(step(select(sim))))	1
step(step(step(select(sim))))	1
step(step(step(repeat(sim))))	1
step(repeat(step(select(sim))))	1
step(repeat(step(step(sim))))	1
step(select(step(sim)))	1
step(select(repeat(sim)))	1

TABLE IV
ALGORITHMS DISCOVERED WHEN APPLIED TO *SUDOKU* WITH A CPU TIME BUDGET

Name	Search Component	Rank	Rank in Table II	Sudoku(4)
Dis#1	select(step(select(step(sim), 0.3)), 0.3)	1	-	**197.2**
Dis#2	step(repeat(step(step(sim)), 10))	2	-	**196.8**
Dis#4	lookahead(select(step(step(sim)), 0.3), 1)	3	-	**196.1**
Dis#5	select(lookahead(step(step(sim)), 1), 0.3)	4	-	**195.9**
Dis#3	lookahead(select(step(step(sim)), 0), 1)	5	-	**195.8**
Dis#6	step(select(step(repeat(sim, 2)), 0.3))	6	-	**195.3**
Dis#9	select(step(step(repeat(sim, 2))), 0)	7	-	**195.2**
Dis#8	step(step(repeat(sim, 2)))	8	-	**194.8**
nmc(2)		9	18	194.7
nmc(3)		10	15	194.5
Dis#7	step(step(select(step(sim), 0)))	10	-	194.5
Dis#10	step(repeat(step(step(sim)), 100))	10	-	194.5
la(1)		13	16	194.2
nmc(4)		14	17	193.7
nmc(5)		15	19	191.4
uct(0.3)		16	4	189.7
uct(0)		17	4	189.4
uct(0.5)		18	6	188.9
uct(1)		19	13	188.8
la(2)		20	20	175.3
la(3)		21	24	170.3
la(4)		22	21	169.3
la(5)		23	23	168.0
is		24	22	167.8

Table IV reports the results we obtain with a budget expressed as a fixed amount of CPU time. For each algorithm, we also indicate its rank in Table II. The new best generic algorithm is now $\mathrm{nmc}(2)$ and eight out of the ten discovered have a better average score than this generic algorithm. In general, we observe that time-based budget favors $\mathrm{nmc}(\cdot)$ algorithms and decreases the rank of $\mathrm{uct}(\cdot)$ algorithms.

In order to better understand the differences between the algorithms found with an evaluations-based budget and those found

TABLE V
SEARCH-COMPONENTS COMPOSITION ANALYSIS

Name	Evaluations-based Budget	Time-based Budget
repeat	8	5
simulate	10	10
select	12	8
step	16	23
lookahead	2	3

TABLE VI
SYMBOLIC REGRESSION TESTBED: TARGET EXPRESSIONS AND DOMAINS

Target Expression $f^P(\cdot)$	Domain
x^3+x^2+x	$[-1,1]$
$x^4+x^3+x^2+x$	$[-1,1]$
$x^5+x^4+x^3+x^2+x$	$[-1,1]$
$x^6+x^5+x^4+x^3+x^2+x$	$[-1,1]$
$\sin(x^2)\cos(x)-1$	$[-1,1]$
$\sin(x)+\sin(x+x^2)$	$[-1,1]$
$\log(x+1)+\log(x^2+1)$	$[0,2]$
$\sqrt{x}$	$[0,4]$

with a time-based budget, we counted the number of occurrences of each of the search components among the ten discovered algorithms in both cases. These counts are reported in Table V. We observe that the time-based budget favors the step search component, while reducing the use of select. This can be explained by the fact that select is our search component that involves the most extra-computational cost, related to the storage and the manipulation of the game tree.

C. Real-Valued Symbolic Regression

Symbolic Regression consists in searching a large space of symbolic expressions for the one that best fits a given regression data set. Usually this problem is treated using GP approaches. Along the lines of [22], we here consider MCS techniques as an interesting alternative to GP. In order to apply MCS techniques, we encode the expressions as sequences of symbols. We adopt the reverse Polish notation (RPN) to avoid the use of parentheses. As an example, sequence $[a, b, +, c, *]$ encodes expression $(a + b) * c$. The alphabet of symbols we used is $\{x, 1, +, -, *, /, \sin, \cos, \log, \exp, \mathrm{stop}\}$. The initial state x_1 is the empty RPN sequence. Each action u then adds one of these symbols to the sequence. When computing the set of valid actions $\mathcal{U}_x$, we reject symbols that lead to invalid RPN sequences, such as $[+, +, +]$. A final state is reached either when the sequence length is equal to a predefined maximum T or when symbol stop is played. In our experiments, we performed the training with a maximal length of $T = 11$. The reward associated to a final state is equal to $1 - \mathrm{mae}$, where mae is the mean absolute error associated to the expression built.

We used a synthetic benchmark, which is classical in the field of GP [23]. To each problem P of this benchmark is associated a target expression $f^P(\cdot) \in \mathbb{R}$, and the aim is to rediscover this target expression, given a finite set of samples $(x, f^P(x))$. Table VI illustrates these target expressions. In each case, we used 20 samples $(x, f^P(x))$, where x was obtained by taking uniformly spaced elements from the indicated domains. The training distribution $\mathcal{D}_P$ was the uniform distribution over the eight problems given in Table VI.

TABLE VII
SYMBOLIC REGRESSION ROBUSTNESS TESTBED:
TARGET EXPRESSIONS AND DOMAINS

Target Expression $f^P(\cdot)$	Domain
x^3-x^2-x	$[-1,1]$
$x^4-x^3-x^2-x$	$[-1,1]$
$x^4+\sin(x)$	$[-1,1]$
$\cos(x^3)+\sin(x+1)$	$[-1,1]$
$\sqrt{(x)}+x^2$	$[0,4]$
x^6+1	$[-1,1]$
$\sin(x^3+x^2)$	$[-1,1]$
$\log(x^3+1)+x$	$[0,2]$

The training budget was $B\ =10\ \ 000$. We evaluate the robustness of the algorithms found in three different ways: by changing the maximal length T from 11 to 21, by increasing the budget B from 10 000 to 100 000, and by testing them on another distribution of problems $\mathcal{D}'_P$. The distribution $\mathcal{D}'_P$ is the uniform distribution over the eight new problems given in Table VII.

The results are shown in Table VIII, where we report directly the mae scores (lower is better). The best generic algorithm is $\mathrm{la}(2)$ and corresponds to one of the discovered algorithms (Dis#3). Five of the discovered algorithms significantly outperform this baseline with scores down to 0.066. Except one of them, all discovered algorithms rely on two nested $\mathrm{lookahead}$ components and generalize in some way the $\mathrm{la}(2)$ algorithm.

When setting the maximal length to $T = 21$, the best generic algorithm is again $\mathrm{la}(2)$, and we have four discovered algorithms that still significantly outperform it. When increasing the testing budget to $B = 100\,000$, nine discovered algorithms out of the ten significantly outperform the best generic algorithms $\mathrm{la}(3)$ and $\mathrm{nmc}(3)$. These results thus show that the algorithms discovered by our approach are robust both with respect to the maximal length T and the budget B.

In our last experiment with the distribution $\mathcal{D}'_P$, there is a single discovered algorithm that significantly outperforms $\mathrm{la}(2)$. However, all ten algorithms still behave reasonably well and significantly better than the nonlookahead generic algorithms. This result is particularly interesting since it shows that our approach was able to discover algorithms that work well for symbolic regression in general, not only for some particular problems.

D. Morpion Solitaire

The classic game of *Morpion Solitaire* [24] is a single-player, pencil and paper game, whose world record has been improved several times over the past few years using MCS techniques [3], [7], [25]. This game is illustrated in Fig. 3. The initial state x_1 is an empty cross of points drawn on the intersections of the grid. Each action places a new point at a grid intersection in such a way that it forms a new line segment connecting consecutive points that include the new one. New lines can be drawn horizontally, vertically, and diagonally. The game is over when no

TABLE VIII
RANKING AND ROBUSTNESS OF THE ALGORITHMS DISCOVERED WHEN APPLIED TO SYMBOLIC REGRESSION

Name	Search Component	Rank	$T = 11$	$T = 21$	$T = 11,\ B = 10^5$	$\mathcal{D}'_P$
Dis#1	step(step(lookahead(lookahead(sim))))	1	***0.066***	***0.083***	***0.036***	0.101
Dis#5	step(repeat(lookahead(lookahead(sim)), 2))	2	***0.069***	***0.085***	***0.037***	0.106
Dis#2	step(lookahead(lookahead(repeat(sim, 2))))	2	***0.069***	***0.084***	***0.038***	**0.100**
Dis#8	step(lookahead(repeat(lookahead(sim), 2)))	2	***0.069***	***0.084***	***0.040***	0.112
Dis#7	step(lookahead(lookahead(select(sim, 1))))	5	***0.070***	0.087	***0.040***	0.103
Dis#6	step(lookahead(lookahead(select(sim, 0))))	6	**0.071**	0.087	***0.039***	0.110
Dis#4	step(lookahead(select(lookahead(sim), 0)))	6	**0.071**	0.087	***0.038***	0.101
Dis#3	step(lookahead(lookahead(sim)))	6	**0.071**	**0.086**	0.056	**0.100**
la(2)		6	0.071	0.086	0.056	0.100
Dis#10	step(lookahead(select(lookahead(sim), 0.3)))	10	0.072	0.088	***0.040***	0.108
la(3)		11	0.073	0.090	0.053	0.101
Dis#9	step(repeat(select(lookahead(sim), 0.3), 5))	12	0.077	0.091	***0.048***	***0.099***
nmc(2)		13	0.081	0.103	0.054	0.109
nmc(3)		14	0.084	0.104	0.053	0.118
la(4)		15	0.088	0.116	0.057	0.101
nmc(4)		16	0.094	0.108	0.059	0.141
la(1)		17	0.098	0.116	0.066	0.119
la(5)		18	0.099	0.124	0.058	0.101
is		19	0.119	0.144	0.087	0.139
nmc(5)		20	0.120	0.124	0.069	0.140
uct(0)		21	0.159	0.135	0.124	0.185
uct(1)		22	0.147	0.118	0.118	0.161
uct(0.3)		23	0.156	0.112	0.135	0.177
uct(0.5)		24	0.153	0.111	0.124	0.184

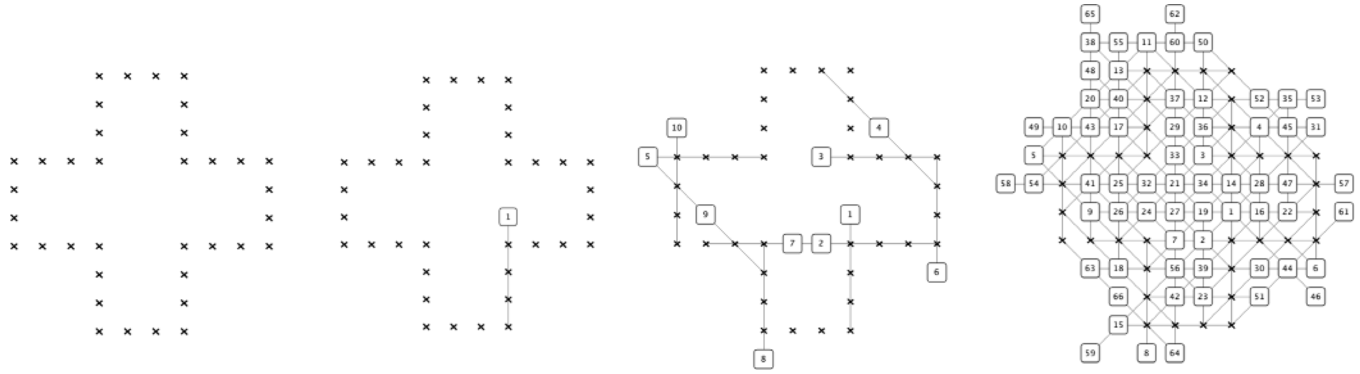


Fig. 3. A random policy that plays the game *Morpion Solitaire* 5T: initial grid; after one move; after ten moves; and game end.

further actions can be taken. The goal of the game is to maximize the number of lines drawn before the game ends, hence the reward associated with the final states is this number.[3]

There exist two variants of the game: "disjoint" and "touching." "Touching" allows parallel lines to share an endpoint, whereas "disjoint" does not. Line segments with different directions are always permitted to share points. The game is NP-hard [26] and presumed to be infinite under certain configurations. In this paper, we treat the $5D$ and $5T$ versions of the game, where 5 is the number of consecutive points to form a line, D means disjoint, and T means touching.

We performed the algorithm discovery in a "single training problem" scenario: the training distribution $\mathcal{D}_P$ always returns the same problem P, corresponding to the $5T$ version of the game. The initial state of P was the one given in the leftmost part of Fig. 3. The training budget was set to $B = 10\,000$. To evaluate the robustness of the algorithms, we, on the one hand, evaluated them on the $5D$ variant of the problem and, on the other hand, changed the evaluation budget from 10 000 to 100 000. The former provides a partial answer to how rule-dependent these algorithms are, while the latter gives insight into the impact of the budget on the algorithms' ranking.

The results of our experiments on *Morpion Solitaire* are given in Table IX. Our approach proves to be particularly successful on this domain: each of the ten discovered algorithms significantly outperforms all tested generic algorithms. Among the generic algorithms, $\mathrm{la}(1)$ gave the best results (90.63), which is 0.46 below the worst of the ten discovered algorithms.

When moving to the $5D$ rules, we observe that all ten discovered algorithms still significantly outperform the best generic algorithm. This is particularly impressive, since it is known that the structure of good solutions strongly differs between the $5D$ and $5T$ versions of the game [24]. The last column of Table IX gives the performance of the algorithms with budget $B = 10^5$. We observe that all ten discovered algorithms also significantly outperform the best generic algorithm in this case. Furthermore, the increase in the budget seems to also increase the gap between the discovered and generic algorithms.

[3] In practice, we normalize this reward by dividing it by 100 to make it approximately fit into the range $[0, 1]$. Thanks to this normalization, we can keep using the same constants for both the UCB policy used in the algorithm discovery and the UCB policy used in select.

TABLE IX
RANKING AND ROBUSTNESS OF ALGORITHMS DISCOVERED WHEN APPLIED TO *MORPION SOLITAIRE*

Name	Search Component	Rank	$5T$	$5D$	$5T, B = 10^5$
Dis#1	step(select(step(simulate),0.5))	1	*91.24*	*63.66*	*97.28*
Dis#4	step(select(step(select(sim,0.5)),0))	2	*91.23*	*63.64*	*96.12*
Dis#3	step(select(step(select(sim,1.0)),0))	3	*91.22*	*63.63*	*96.02*
Dis#2	step(step(select(sim,0)))	4	*91.18*	*63.63*	*96.78*
Dis#8	step(select(step(step(sim)),1))	5	*91.12*	*63.63*	*96.67*
Dis#9	step(select(step(select(sim,0)),0.3))	6	*91.22*	*63.67*	*96.02*
Dis#5	select(step(select(step(sim),1.0)),0)	7	*91.16*	*63.65*	*95.79*
Dis#10	step(select(step(select(sim,1.0)),0.0))	8	*91.21*	*63.62*	*95.99*
Dis#6	lookahead(step(step(sim)))	9	*91.15*	*63.68*	*96.41*
Dis#7	lookahead(step(step(select(sim, 0))))	10	*91.08*	63.67	*96.31*
la(1)		11	90.63	63.41	95.09
nmc(3)		12	90.61	63.44	95.59
nmc(2)		13	90.58	63.47	94.98
nmc(4)		14	90.57	63.43	95.24
nmc(5)		15	90.53	63.42	95.17
uct(0)		16	89.40	63.02	92.65
uct(0.5)		17	89.19	62.91	92.21
uct(1)		18	89.11	63.12	92.83
uct(0.3)		19	88.99	63.03	92.32
la(2)		20	85.99	62.67	94.54
la(3)		21	85.29	61.52	89.56
is		21	85.28	61.40	88.83
la(4)		23	85.27	61.53	88.12
mcts		24	85.26	61.48	89.46
la(5)		25	85.12	61.52	87.69

E. Discussion

We have seen that on each of our three testbeds, we discovered algorithms that are competitive with, or even significantly better than, generic ones. This demonstrates that our approach is able to generate new MCS algorithms specifically tailored to the given class of problems. We have performed a study of the robustness of these algorithms by either changing the problem distribution or by varying the budget B, and found that the algorithms discovered can outperform generic algorithms even on problems significantly different from those used for the training.

The importance of each component of the grammar depends heavily on the problem. For instance, in *Symbolic Regression*, all ten best algorithms discovered rely on two nested $\mathrm{lookahead}$ components, whereas in *Sudoku* and *Morpion Solitaire*, step and select appear in the majority of the best algorithms discovered.

VI. Related Work

Methods for automatically discovering MCS algorithms can be characterized through three main components: the space of candidate algorithms, the performance criterion, and the search method for finding the best element in the space of candidate algorithms.

Usually, researchers consider spaces of candidate algorithms that only differ in the values of their constants. In such a context, the problem amounts to tuning the constants of a generic MCS algorithm. Most of the research related to the tuning of these constants takes, as a performance criterion, the mean score of the algorithm over the distribution of target problems. Many search algorithms have been proposed for computing the best constants. For instance, Perrick *et al.* [27] employ a grid search approach combined with self-playing, Chaslot *et al.* [28] use cross entropy as a search method to tune an agent playing *Go*, Coulom [29] presents a generic black-box optimization method based on local quadratic regression, Maes *et al.* [30] use estimation distribution algorithms with Gaussian distributions, Chapelle and Li [31] use Thompson sampling, and Bourki *et al.* [32] uses, as in the present paper, a multi-armed bandit approach. The paper [33] studies the influence of the tuning of MCS algorithms on their asymptotic consistency and shows that pathological behavior may occur with tuning. It also proposes a tuning method to avoid such behavior.

Research papers that have reported empirical evaluations of several MCS algorithms in order to find the best one are also related to this automatic discovery problem. The space of candidate algorithms in such cases is the set of algorithms they compare, and the search method is an exhaustive search procedure. As a few examples, Perrick *et al.* [27] report on a comparison between algorithms that differ in their selection policy, Gelly and Silver [34] and Chaslot *et al.* [35] compare improvements of the UCT algorithm (RAVE and progressive bias) with the original one on the game of *Go*, and St-Pierre *et al.* [36] evaluate different versions of a two-player MCS algorithm on generic sparse bandit problems. Browne *et al.* [37] provide an in-depth review of different MCS algorithms and their successes in different applications.

The main feature of the approach proposed in this paper is that it builds the space of candidate algorithms by using a rich grammar over the search components. In this sense, references [38] and [39] are certainly the papers which are the closest to ours, since they also use a grammar to define a search space, for, respectively, two-player games and multiarmed bandit problems. However, in both cases, this grammar only models a selection policy and is made of classic functions such as $+$, $-$, $*$, $/$, $\log$, $\exp$, and $\sqrt{}$. We have taken one step forward, by directly defining a grammar over the MCS algorithms that covers very different MCS techniques. Note that the search technique of [38] is based on GP.

The decision as to what to use as the performance criterion is not as trivial as it looks, especially for multiplayer games, where opponent modeling is crucial for improving over game-theoretically optimal play [40]. For example, the maximization of the victory rate or loss minimization against a wide variety of opponents for a specific game can lead to different choices of algorithms. Other examples of criteria to discriminate between algorithms are simple regret [32] and the expected performance over a distribution density [41].

VII. Conclusion

In this paper, we have addressed the problem of automatically identifying new MCS algorithms that perform well on a distribution of training problems. To do so, we introduced a grammar over the MCS algorithms that generates a rich space of candidate algorithms (and which describes, along the way, using a particularly compact and elegant description, several well-known MCS algorithms). To efficiently search inside this space of candidate algorithms for the one(s) having the best average performance on the training problems, we relied on a multiarmed bandit type of the optimization algorithm.

Our approach was tested on three different domains: *Sudoku*, *Morpion Solitaire*, and *Symbolic Regression*. The results showed that the algorithms discovered this way often significantly outperform generic algorithms such as UCT or NMC.

Moreover, we showed that they had good robustness properties, by changing the testing budget and/or by using a testing problem distribution different from the training distribution.

This work can be extended in several ways. For the time being, we used the mean performance over a set of training problems to discriminate between different candidate algorithms. One direction for future work would be to adapt our general approach to use other criteria, e.g., worst case performance measures. In its current form, our grammar only allows using predefined simulation policies. Since the simulation policy typically has a major impact on the performance of an MCS algorithm, it could be interesting to extend our grammar so that it could also "generate" new simulation policies. This could be arranged by adding a set of simulation policy generators in the spirit of our current search-component generators. Previous work has also demonstrated that the choice of the selection policy could have a major impact on the performance of Monte Carlo tree search algorithms. Automatically generating selection policies is thus also a direction for future work. Of course, working with richer grammars will lead to larger candidate algorithm spaces, which, in turn, may require developing more efficient search methods than the multiarmed bandit one used in this paper. Finally, another important direction for future research is to extend our approach to more general settings than single-player games with full observability.

REFERENCES

[1] L. Kocsis and C. Szepesvári, "Bandit based Monte Carlo planning," in *Proc. 17th Eur. Conf. Mach. Learn.*, 2006, pp. 282–293.

[2] R. Coulom, "Efficient selectivity and backup operators in Monte Carlo tree search," in *Proc. 5th Int. Conf. Comput. Games*, Turin, Italy, 2006, pp. 72–83.

[3] T. Cazenave, "Nested Monte Carlo search," in *Proc. 21st Int. Joint Conf. Artif. Intell.*, 2009, pp. 456–461.

[4] J. Méhat and T. Cazenave, "Combining UCT and nested Monte Carlo search for single-player general game playing," *IEEE Trans. Comput. Intell. AI Games*, vol. 2, no. 4, pp. 271–277, Dec. 2010.

[5] G. Tesauro and G. R. Galperin, "On-line policy improvement using Monte Carlo search," *Proc. Neural Inf. Process. Syst.*, vol. 96, pp. 1068–1074, 1996.

[6] P. Auer, P. Fischer, and N. Cesa-Bianchi, "Finite-time analysis of the multi-armed bandit problem," *Mach. Learn.*, vol. 47, pp. 235–256, 2002.

[7] T. Cazenave, "Reflexive Monte Carlo search," in *Proc. Comput. Games Workshop*, Amsterdam, The Netherlands, 2007, pp. 165–173.

[8] G. Chaslot, S. de Jong, J.-T. Saito, and J. Uiterwijk, "Monte-Carlo tree search in production management problems," in *Proc. Benelux Conf. Artif. Intell.*, 2006, pp. 91–98.

[9] M. P. D. Schadd, M. H. M. Winands, H. J. V. D. Herik, G. M. J. b. Chaslot, and J. W. H. M. Uiterwijk, "Single-player Monte-Carlo tree search," in *Proceedings of Computers and Games (CG)*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2008, vol. 5131, pp. 1–12.

[10] F. de Mesmay, A. Rimmel, Y. Voronenko, and M. Püschel, "Bandit-based optimization on graphs with application to library performance tuning," in *Proc. Int. Conf. Mach. Learn.*, Montréal, QC, Canada, 2009, pp. 729–736.

[11] G.-B. Chaslot, J.-B. Hoock, J. Perez, A. Rimmel, O. Teytaud, and M. Winands, "Meta Monte Carlo tree search for automatic opening book generation," in *Proc. IJCAI Workshop Gen. Intell. Game Playing Agents*, 2009, pp. 7–12.

[12] F. Maes, L. Wehenkel, and D. Ernst, "Learning to play K-armed bandit problems," in *Proc. Int. Conf. Agents Artif. Intell.*, Vilamoura, Algarve, Portugal, Feb. 2012.

[13] F. Maes, L. Wehenkel, and D. Ernst, "Meta-learning of exploration/exploitation strategies: The multi-armed bandit case," in *Proc. Int. Conf. Agents Artif. Intell.*, 2012 [Online]. Available: arXiv:1207.5208

[14] M. Castronovo, F. Maes, R. Fonteneau, and D. Ernst, "Learning exploration/exploitation strategies for single trajectory reinforcement learning," in *Proc. 10th Eur. Workshop Reinforcement Learn.*, Edinburgh, Scotland, Jun. 2012.

[15] F. Maes, R. Fonteneau, L. Wehenkel, and D. Ernst, "Policy search in a space of simple closed-form formulas: Towards interpretability of reinforcement learning," in *Discovery Science*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, Oct. 2012, vol. 7569, pp. 37–51.

[16] S. Bubeck, R. Munos, and G. Stoltz, "Pure exploration in multi-armed bandits problems," in *Proc. Algorithmic Learn. Theory*, 2009, pp. 23–37.

[17] J. Koza and R. Poli, "Genetic programming," in *Proc. Search Methodol.*, E. K. Burke and G. Kendall, Eds., 2005, pp. 127–164.

[18] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Mach. Learn.*, vol. 47, no. 2, pp. 235–256, 2002.

[19] P.-A. Coquelin and R. Munos, "Bandit algorithms for tree search," in *Proc. Uncertainty Artif. Intell.*, Vancouver, BC, Canada, 2007, arXiv preprint cs/0703062.

[20] J.-B. Hoock and O. Teytaud, "Bandit-based genetic programming," in *Proc. 13th Eur. Conf. Genetic Programm.*, 2010, pp. 268–277.

[21] Z. Geem, "Harmony search algorithm for solving Sudoku," in *Proc. Int. Conf. Knowl.-Based Intell. Inf. Eng. Syst.*, 2007, pp. 371–378.

[22] T. Cazenave, "Nested Monte Carlo expression discovery," in *Proc. Eur. Conf. Artif. Intell.*, Lisbon, Portugal, 2010, pp. 1057–1058.

[23] N. Uy, N. Hoai, M. O'Neill, R. McKay, and E. Galván-López, "Semantically-based crossover in genetic programming: Application to real-valued symbolic regression," *Genetic Programm. Evolvable Mach.*, vol. 12, no. 2, pp. 91–119, 2011.

[24] C. Boyer, 2012 [Online]. Available: http://www.morpionsolitaire.com

[25] C. Rosin, "Nested rollout policy adaptation for Monte Carlo tree search," in *Proc. 22nd Int. Joint Conf. Artif. Intell.*, Barcelona, Spain, 2011, pp. 649–654.

[26] E. Demaine, M. Demaine, A. Langerman, and S. Langerman, "Morpion solitaire," *Theory Comput. Syst.*, vol. 39, no. 3, pp. 439–453, 2006.

[27] P. Perrick, D. St-Pierre, F. Maes, and D. Ernst, "Comparison of different selection strategies in Monte Carlo tree search for the game of Tron," in *Proc. IEEE Conf. Comput. Intell. Games*, Granada, Spain, 2012, pp. 242–249.

[28] I. Chaslot, M. Winands, and H. van den Herik, "Parameter tuning by the cross-entropy method," in *Proc. Eur. Workshop Reinforcement Learn.*, 2008.

[29] R. Coulom, "CLOP: Confident local optimization for noisy black-box parameter tuning," in *Advances in Computer Games*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2012, vol. 7168, pp. 146–157.

[30] F. Maes, L. Wehenkel, and D. Ernst, "Optimized look-ahead tree search policies," in *Recent Advances in Reinforcement Learning*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2012, vol. 7188, pp. 189–200.

[31] O. Chapelle and L. Li, "An empirical evaluation of Thompson sampling," in *Proc. Neural Inf. Process. Syst.*, 2011, pp. 2249–2257.

[32] A. Bourki, M. Coulm, P. Rolet, O. Teytaud, and P. Vayssiere, "Parameter tuning by simple regret algorithms and multiple simultaneous hypothesis testing," in *Proc. Int. Conf. Inf. Control Autom. Robot.*, 2010.

[33] V. Berthier, H. Doghmen, and O. Teytaud, "Consistency modifications for automatically tuned Monte Carlo tree search," in *Proc. 4th Conf. Learn. Intell. Optim.*, 2010, pp. 111–124.

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

[35] G. Chaslot, M. Winands, H. Herik, J. Uiterwijk, and B. Bouzy, "Progressive strategies for Monte Carlo tree search," in *Proc. 10th Joint Conf. Inf. Sci.*, 2007, pp. 655–661.

[36] D. St-Pierre, Q. Louveaux, and O. Teytaud, "Online sparse bandit for card game," in *Advances in Computer Games*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2012, vol. 7168, pp. 295–305.

[37] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012.

[38] T. Cazenave, "Evolving Monte Carlo tree search algorithms," Tech. Rep., 2007.

[39] F. Maes, L. Wehenkel, and D. Ernst, "Automatic discovery of ranking formulas for playing with multi-armed bandits," in *Proc. 9th Eur. Workshop Reinforcement Learn.*, 2011, pp. 5–17.

[40] D. Billings, A. Davidson, J. Schaeffer, and D. Szafron, "The challenge of poker," *Artif. Intell.*, vol. 134, no. 1–2, pp. 201–240, 2002.

[41] V. Nannen and A. Eiben, "Relevance estimation and value calibration of evolutionary algorithm parameters," in *Proc. 20th Int. Joint Conf. Artif. Intell.*, 2007, pp. 975–980.

Francis Maes received the Ph.D. degree in computer science from the University Pierre et Marie Curie, Paris 6, France, in 2009.

He was a Postdoctoral Fellow in the Systems and Modeling Research Unit, University of Liège, Liège, Belgium, until July 2012. He is now a Postdoctoral Fellow in the Declarative Languages and Artificial Intelligence Research Group, Catholic University of Leuven (KULeuven), Leuven, Belgium. His research interests include machine learning with structured data, reinforcement learning, and meta-learning.

David Lupien St-Pierre received the B.Sc. and M.Sc. (in electrical engineering) degrees from the Université du Québec à Trois-Rivières, Trois-Rivières, QC, Canada, in 2005 and 2007, respectively, and the M.Sc. degree in computer science from the University of Glasgow, Glasgow, U.K., in 2009. He is currently working toward the Ph.D. degree in computer engineering at the University of Liège, Liège, Belgium.

His research interests include artificial intelligence, game theory, and machine learning.

Damien Ernst received the M.S. and Ph.D. degrees in engineering from the University of Liège, Liège, Belgium, in 1998 and 2003, respectively.

He is currently an Associate Professor at the University of Liège, where he is affiliated with the Systems and Modeling Research Unit. He is also the holder of the EDF-Luminus Chair on Smart Grids. His research interests include power system control and reinforcement learning.