Appears in: Proc. of IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL'13), Singapore, April 16-19, 2013

Optimized Look-Ahead Trees: Extensions to Large and Continuous Action Spaces

Tobias Jung
Institut Montefiore
University of Liège, Belgium
Email: tjung@ulg.ac.be

Damien Ernst
Institut Montefiore
University of Liège, Belgium
Email: dernst@ulg.ac.be

Francis Maes
Department of Computer Science
Katholieke Universiteit Leuven, Belgium
Email: francis.maes@cs.kuleuven.be

***Abstract*—This paper studies look-ahead tree based control policies from the viewpoint of online decision making with constraints on the computational budget allowed per decision (expressed as number of calls to the generative model). We consider optimized look-ahead tree (OLT) policies, a recently introduced family of hybrid techniques, which combine the advantages of look-ahead trees (high precision) with the advantages of direct policy search (low online cost) and which are specifically designed for limited online budgets. We present two extensions of the basic OLT algorithm that on the one side allow tackling deterministic optimal control problems with large and continuous action spaces and that on the other side can also help to further reduce the online complexity.**

I. Introduction

This paper addresses the computational problem of determining a (near-) optimal policy for deterministic optimal control tasks with potentially high-dimensional state and large or continuous action spaces, provided that a simulation model of the dynamics exists and is given *a priori*, and by making explicit the constraint that in real world applications only limited computational resources are available. The particular technique (or better, family of techniques) we are going to study is called *optimized look-ahead tree policies* (OLT) [7], [9]. OLT is a model-based hybrid technique that bridges the gap between look-ahead tree policies (LT) [3], [6] and direct policy search (DPS). Like LT policies, every time the control policy is requested to output an action for the current system state, OLT develops non-uniformly a look-ahead tree until a computational budget is exhausted and then returns the first action that lies on the most promising path. Unlike LT policies, OLT does not use generic (i.e., task-independent) heuristics to determine how the tree is developed and how the paths are scored. Instead, it parameterizes these heuristics and optimizes the parameters *for each specific task and given computational budget* in an offline learning phase via black-box global optimization, thus turning OLT effectively into a variant of standard DPS with a non-standard form of implicit policy representation.

OLT was first introduced in [9], with the more comprehensive study in [7] forming the basis of the present work. Here, we extend the earlier approach in two ways:

- **Conceptual:** we make explicit the computational cost involved in the whole learning procedure by considering separately the offline complexity (effort required to learn the policy) and online complexity (effort required during runtime to calculate decisions from the policy).
- **Algorithmic:** the original OLT/LT algorithm was feasible only for action spaces with a finite and small number of actions. Here, we will present two different extensions aimed at eliminating this limitation and allowing OLT to be applied to problems with large or continuous action spaces.

The paper is structured as follows. Section II formalizes the problem we address and Section III provides an overview of optimized look-ahead trees. We then introduce two new extensions of OLT to deal with large and/or continuous action spaces: Section IV presents OLTs with action-sampling and Section V presents OLTs with recursive action splitting. Section VI provides an experimental evaluation. Section VII overviews related work and finally Section VIII concludes.

II. Problem Statement

We begin by formally stating the class of optimal control problems this paper is about:

A. Model-based optimal control problems

We consider a discrete-time system whose dynamics are given by $x_{t+1} = f(x_t, a_t)$, where x indicates an element of some state space X and a indicates an element of some action space A. We assume, largely for simplicity and to facilitate running the typical simulation experiments consisting of vector-valued domains, that both X and A are continuous vector spaces, i.e., $X \subset \mathbb{R}^{n_x}$, $A \subset \mathbb{R}^{n_u}$ (the presented method should also work with some modifications for other forms of X and A). The system evolves at discrete time steps with $t = 1, 2, \ldots$; for every transition we make, we observe a scalar reward $\varrho(x_t, a_t)$ which serves as the performance measure to optimize over time. We assume that the transition dynamics f and ϱ are available during learning, i.e., our presented approach is *model-based*.

Let $\pi : X \to A$ denote a stationary deterministic policy, i.e., a mapping from states to actions. For any given policy π and state x_0, the infinite horizon discounted sum of rewards $V^\pi(x_0)$ is defined as

$$V^\pi(x_0) := \lim_{T\to\infty} \sum_{t=0}^{T} \gamma^t \varrho(x_t, \pi(x_t)) \qquad (1)$$

where $x_{t+1} = f(x_t, \pi(x_t))$ and where $0 < \gamma < 1$ is a discount factor. The optimal value function is defined as the maximum over all policies, $V^*(x_0) := \max_\pi V^\pi(x_0)$, and satisfies the discrete-time Hamilton-Jacobi-Bellman (HJB) equation $V^*(x) = \max_a \left[\varrho(x,a) + \gamma V^*\big(f(x,a)\big)\right]$ for all $x \in X$. If we are able to determine V^*, the optimal action for every state x can be derived as $\pi^*(x) = \operatorname{argmax}_a \left[\varrho(x,a) + \gamma V^*\big(f(x,a)\big)\right]$. However, as is well known, solving the HJB equation exactly is possible only in some cases (e.g., LQR); in the general nonlinear case and for higher dimensional spaces this presents an open research problem.

B. Direct policy search

Direct policy search (DPS) approaches do not try to estimate V^π or V^*, but solve the conceptually simpler problem of finding a policy that "works well" for a range of initial conditions by directly optimizing performance in policy space. In our case, we define this objective as follows: given a set of initial states $X_0 \subset X$, our goal is to find a policy that maximizes the performance over the states in X_0, i.e., we want to find $\operatorname{argmax}_\pi \sum_{x_0 \in X_0} V^\pi(x_0)$ (or $\operatorname{argmax}_\pi \sum_{x_0 \in X_0} p(x_0) V^\pi(x_0)$, where $p(x_0) \geq 0$ are some weights).

In general DPS, one assumes that policies are functions $\pi_\theta := \pi(\cdot\,; \theta)$ parameterized by some vector $\theta \in \mathbb{R}^d$. The optimization over policies is thus turned into an optimization over real-valued vectors

$$\operatorname*{argmax}_{\theta \in \mathbb{R}^d} V_{X_0}(\theta) := \sum_{x_0 \in X_0} V^{\pi_\theta}(x_0). \tag{2}$$

Given a vector θ, the objective function V_{X_0} can be evaluated by simulating the system under the policy π_θ from all the states in X_0 and summing all the rewards obtained (recall that we assume a model (f, ϱ) is available for simulating the system). In practice, to avoid the infinite sum in Eq. (1), we have to truncate the infinite horizon to a (typically large) number of finite simulation steps H. The objective function in Eq. (2) is thus effectively replaced by

$$\operatorname*{argmax}_{\theta \in \mathbb{R}^d} V_{X_0}(\theta) := \sum_{x_0 \in X_0} \sum_{t=0}^{H} \gamma^t \varrho\big(x_t, \pi(x_t;\theta)\big), \tag{3}$$

where $\forall t: \; x_{t+1} = f(x_t, \pi(x_t;\theta))$.

C. Goal: constrained online budget

One important feature of learning algorithms that often gets overlooked when making comparisons is their computational cost. We believe it to be worthwile to examine their computational cost from two points of view:

- **Offline complexity**, which we understand as the effort required to obtain the policy (once we are told f, ϱ)
- **Online complexity**, which we understand as the effort required at each step to calculate the best decision (once we have found π_θ).

Having made this distinction, we can characterize DPS and LT as lying at opposite ends of the offline complexity / online complexity spectrum [7]. DPS techniques typically require huge offline resources for two reasons. First, one has to select an appropriate space of parameterized policies that works well for the targeted problem, which usually implies trial and error experimentation with lots of manual tuning and tweaking. Second, once a suitable parameterization is found, the parameters θ have to be learned via global optimization, which can be expensive, especially when the number of parameters is large. However, once learning is done, executing the policy is very fast and thus requires only negligible online computational resources. LT policies on the other hand are the exact opposite. LT policies do not require any offline efforts: they can be applied to almost any optimal control problem without any prior knowledge and without any prior computation (this even without depending on the dimensionality of the state space). However, LT policies typically require a huge amount of online resources in order to grow trees of sufficient depth.

	Performance	Online cost	Offline cost
DPS	good – very good	very low	very high
LT	very good	very high	zero
OLT	very good	low	medium – high

Fig. 1. Comparison chart of the DPS, LT, OLT family of algorithms along the criteria *performance* (how well does the resulting policy perform in a given task), *online cost* (how much does calculating each decision cost), and *offline cost* (how much does calculating the policy cost).

OLT is a hybrid between LT and DPS and combines the advantages of LT (potentially very good performance, no hassles in finding the right policy representation, largely independent of the dimensionality of the state space) with the advantages of DPS (low cost during runtime). Figure 1 summarizes these essential characteristics in a small comparison chart.

Unlike traditional DPS techniques that rely on parametric function approximators such as neural networks to directly compute control actions for a given state, the computation of an OLT policy involves a non-trivial algorithm which depends on both the parameters θ and on the model (f, ϱ). To emphasize this distinction, we will write $\pi_{f,\varrho}(\cdot\,;\theta)$ whenever we refer explicitly to an OLT policy. Moreover, the OLT family of algorithms was specifically designed to deal with the situation of a *constrained online budget*, i.e., it is an answer to the following question:

> "Assume we have a model of a system (f, ϱ). What is the best policy $\pi_{f,\varrho}(\cdot\,;\theta)$ one can find if the policy is allowed to spend no more than K resources to output a decision at each step (and in addition we have no more than M resources for offline optimization)."

In general, several possibilities would exist to measure the online cost, such as CPU time or number of operations. In practice, for complex real-world problems the most expensive step will be evaluating f (simulating a transition). Therefore we will define the online complexity of a particular policy $\pi_{f,\varrho}(\cdot\,;\theta)$ as *the number of calls to the model* f it makes each time an action is calculated.

III. Optimized look-ahead trees

A. Basics

All forms of policies we consider in this paper work by using the same general strategy. Assume that at time t the current state of the system is x_t. In order to determine an action $\pi_{f,\varrho}(x_t)$, we use the model (f, ϱ) to develop non-uniformly a look-ahead tree starting from state x_t until the *a priori* specified online budget of allowed node expansions (i.e., calls of the expensive transition function f) is exhausted. The look-ahead tree is stored as a list of open nodes (i.e., leaf nodes whose successors have not yet been expanded) where each node is a compound structure consisting of the associated state and the discounted sum of rewards obtained along the current path (plus some other information). Having built the tree, we then return the first action on the path from the root to the most promising leaf node. Once this action has been determined, the tree is discarded, the action is performed, whereupon the state of the system changes to $x_{t+1} = f(x_t, \pi_{f,\varrho}(x_t))$ and we start again by building a new tree from x_{t+1}.

This general strategy depends on two abstract[1] functions which will be instantiated in various ways to produce the different algorithms we propose: the *node expansion heuristic*, which determines which node to expand next; and the *action-selection heuristic*, which decides which leaf node is the most promising one and hence which action to select once the tree is built.

A second axis along which we will vary this general strategy is whether we define the nodes over states or state-action pairs:

- **Look-ahead trees over states:** nodes are defined over states. Each time we expand a node, we generate for each action the successor state and add a corresponding new leaf node to the list. Note that doing this assumes that the number of actions $nAct$ the decision maker has is finite and small. Each node expansion adds $nAct$ new nodes to the list and costs $nAct$ evaluations of the model (f, ϱ).
- **Look-ahead trees over state-actions:** nodes are defined over state-action pairs. Each time we expand a node, we only generate the one successor state that corresponds to the action prescribed by the node. For this successor state, we then add for each action one new node to the list of open nodes. Thus each node expansion adds, as before, $nAct$ new nodes to the list; however, it only costs one evaluation of the model (f, ϱ).

While it would seem that look-ahead trees over state-actions are potentially much more efficient than those defined over only states, the chief problem is that they require the node expansion heuristic to be defined in a meaningful way for both states and actions (such that different actions from the same state result in different scores).

[1]We use the term "abstract" on purpose to emphasize the analogy with programming languages. Both the node expansion heuristic and the action selection heuristic are abstract in the sense that they fulfill a specified role, namely mapping nodes to scores, but leave open the details of how exactly this mapping is performed algorithmically.

B. Notation and illustration

To make things more clear, let us introduce some notation and give a small example. Let $\mathcal{T}$ be the list of open nodes. Each node $\texttt{n} \in \mathcal{T}$ represents the outcome of applying a particular sequence of actions, starting from x_t. Each node is a `struct` object of type $\aleph$ with the following members: $\texttt{n}.x$ (the associated state), $\texttt{n}.d$ (the depth of $\texttt{n}$ with respect to the root), $\texttt{n}.r$ (the cumulative discounted reward obtained along the path from the root to $\texttt{n}$), $\texttt{n}.\pi$ (the first action on the path) and $\texttt{n}.\mathrm{exp\_score}$ (the expansion score, which results from applying the node expansion heuristic to $\texttt{n}$). Let $\mathrm{exp\_score} : \aleph \longrightarrow \mathbb{R}$ denote the abstract node expansion heuristic and $\mathrm{act\_score} : \aleph \longrightarrow \mathbb{R}$ the abstract action-selection heuristic. Both heuristics will use the information stored in $\texttt{n}.x, \texttt{n}.r, \texttt{n}.d$ to compute the respective score for the node. If the node is defined over state-action pairs, it has the additional member $\texttt{n}.a$, which denotes the action that is applied when the node gets expanded.

In Figure 2, we illustrate for two actions a_1 and a_2 the basic working of decision making via look-ahead trees defined over states and state-actions. For clarity, we also summarize in Figure 3 in pseudo-code the necessary computational steps.

C. Look-ahead tree policies

LT policies were introduced in [6] and are intended for deterministic problems with finite and small action spaces. With LT policies, nodes are defined over states. The node expansion heuristic is instantiated by

$$\mathrm{exp\_score}(\texttt{n}) = \texttt{n}.r + \gamma^{\texttt{n}.d}\overline{B}/(1-\gamma) \tag{4}$$

where $\overline{B}$ the maximum attainable reward in the domain, and γ the discount factor from Eq. (1). The action-selection heuristic is instantiated by

$$\mathrm{act\_score}(\texttt{n}) = \texttt{n}.r + \gamma^{\texttt{n}.d}\underline{B}/(1-\gamma) \tag{5}$$

where $\underline{B}$ the minimum attainable reward in the domain. Note that both scores form upper and lower bounds on $V^*(x_t)$.

D. Optimized look-ahead tree policies

OLT policies as described in [7], [9] are also intended for deterministic problems with finite and small action spaces, with the nodes being also defined over states. However, OLT policies differ from LT ones in one significant way: whereas LT uses a generic node expansion heuristic, OLT relies on a parameterized node expansion heuristic $\mathrm{exp\_score}(\texttt{n}; \theta)$ where the parameters θ are specifically optimized in an offline learning phase for the given target domain (f, ϱ). The main advantage of OLT over LT is that this optimization can lead to a substantial reduction of the number of node expansions necessary to output good control actions (as was empirically demonstrated in [7], [9]), meaning that OLT can achieve the same performance as LT at a significantly lower online cost. (The disadvantage of OLT is of course that it needs this prior offline learning.) Moreover, OLT is automatically optimized for a given budget of node expansions, making it the ideal choice for settings where constraints on the online complexity

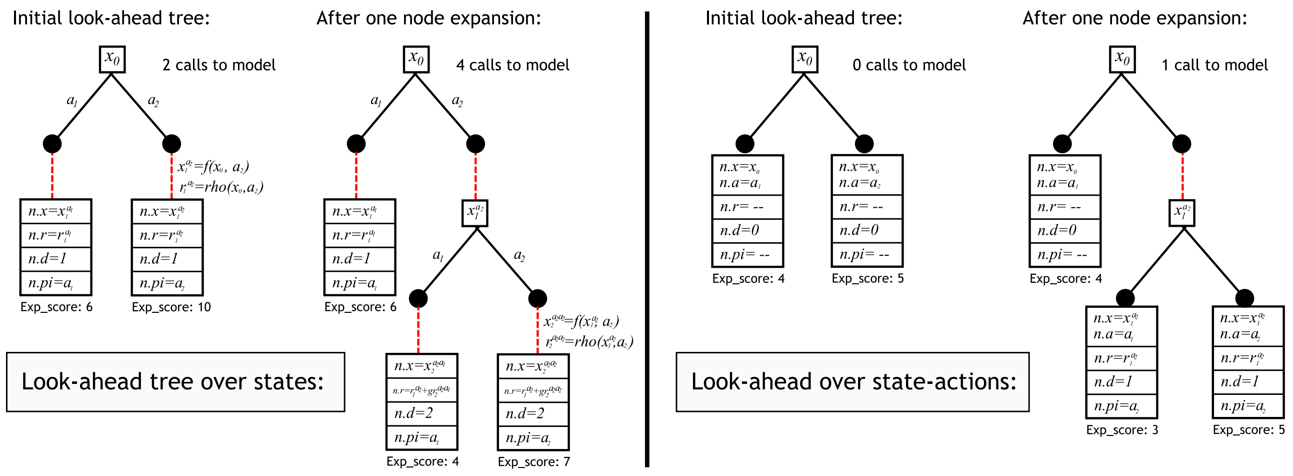


Fig. 2. Illustrating look-ahead trees defined over nodes respresenting states (left side) and nodes representing state-action pairs (right side) for an example with two actions a_1 and a_2. The dotted lines indicate for each case at what point calls to the generative model (f, ϱ) occur. Each node with an exp_score is an element $\mathtt{n}$ in the list of open nodes $\mathcal{T}$. See the main text for a detailed description of the other contents of a node.

Input: state x, weights θ
Output: policy action $\pi_{f,\varrho}(x;\theta)$

Depends on:

f	transition function
ϱ	reward function
γ	discount factor
K	online budget (allowed calls to model (f, ϱ))

Node n is a `struct` of type $\aleph$ consisting of fields:

$n.x$	state
$(n.a$	action)
$n.d$	depth
$n.r$	cumulative discounted reward on path
$n.\pi$	first action on path
$n.$exp_score	node expansion score

```
/* OLT over states */
$x_0 = x$, $\mathcal{T} = \emptyset$, $k = 0$

1. Initialize
    For $i = 1, \ldots, nAct$
        Make transition $(f, \varrho)$ from $x_0$:
            $n.x = f(x_0, a_i)$
            $n.r = \varrho(x_0, a_i)$
        $n.d = 1$
        $n.\pi = a_i$
        $n.\text{exp\_score} = \text{exp\_score}(n;\theta)$
        Add $n$ to the list of open nodes $\mathcal{T}$
    End
2. Main loop
    While $k < K$
        Find node with highest expansion score:
            $n^* := \text{argmax}_{n\in\mathcal{T}}\ n.\text{exp\_score}$
        For $i = 1, \ldots, nAct$
            Make transition $(f, \varrho)$ from $n^*.x$:
                $n.x = f(n^*.x, a_i)$
                $n.r = n^*.r + \gamma^{n^*.d}\varrho(n^*.x, a_i)$
                $k = k + 1$
            $n.d = n^*.d + 1$
            $n.\pi = n^*.\pi$
            $n.\text{exp\_score} = \text{exp\_score}(n;\theta)$
            Add $n$ to the list of open nodes $\mathcal{T}$
        End
        Remove $n^*$ from $\mathcal{T}$
    End

3. Return best action
    Find node with highest action selection score:
        $n^* = \text{argmax}_{n\in\mathcal{T}}\ \text{act\_score}(n;\theta)$
    Return $\pi_{f,\varrho}(x_t;\theta) := n^*.\pi$
```

```
/* OLT over state-actions */
$x_0 = x$, $\mathcal{T} = \emptyset$, $k = 0$

1. Initialize
    For $i = 1, \ldots, nAct$
        $n.x = x_0$
        $n.a = a_i$
        $n.r = 0$
        $n.d = 0$
        $n.\pi = a_i$
        $n.\text{exp\_score} = \text{exp\_score}(n;\theta)$
        Add $n$ to the list of open nodes $\mathcal{T}$
    End
2. Main loop
    While $k < K$
        Find node with highest expansion score:
            $n^* := \text{argmax}_{n\in\mathcal{T}}\ n.\text{exp\_score}$
        Make transition $(f, \varrho)$ from $n^*.x, n^*.a$:
            $n.x = f(n^*.x, n^*.a)$
            $n.r = n^*.r + \gamma^{n^*.d}\varrho(n^*.x, n^*.a)$
            $k = k + 1$
        For $i = 1, \ldots, nAct$
            $n.a = a_i$
            $n.d = n^*.d + 1$
            $n.\pi = n^*.\pi$
            $n.\text{exp\_score} = \text{exp\_score}(n;\theta)$
            Add $n$ to the list of open nodes $\mathcal{T}$
        End
        Remove $n^*$ from $\mathcal{T}$
    End

3. Return best action
    Find node with highest action selection score:
        $n^* = \text{argmax}_{n\in\mathcal{T}}\ \text{act\_score}(n;\theta)$
    Return $\pi_{f,\varrho}(x_t;\theta) := n^*.\pi$
```

Fig. 3. The general form of all LT/OLT policies. Specific variants of the general algorithm are obtained by implementing the two abstract functions exp_score : $\aleph \longrightarrow \mathbb{R}$ and act_score : $\aleph \longrightarrow \mathbb{R}$, which may or may not depend on additional parameters θ.

exist and are known in advance during the design of the controller.

The work in [7] used the following specific parameterization

$$\text{exp\_score}(n;\theta)=\sum_{i=1}^{n_x}\mathtt{n}.x_i\cdot\big(\theta_i+\mathtt{n}.d\cdot\theta_{i+n_x}+\mathtt{n}.r\cdot\theta_{i+2n_x}\big) \quad (6)$$

where $\theta \in \mathbb{R}^{3n_x}$ is the parameter vector and n_x the dimensionality of the state space. Of course, other forms of parameterizations would also be possible (e.g., neural networks), but were not explored in the original work. Similarly, one could also imagine parameterizing act_score in an analogous manner and jointly optimizing over both sets of parameters (this again was not explored in [7], which used as action-selection heuristic the one given for LT in Eq. (5)). No matter how the heuristics are parameterized, the "best" setting of the parameter vector θ is determined by solving Eq. (3) via black-box global optimization, with e.g. an estimation of distribution algorithm or Gaussian process optimization as in [7].

IV. OLTs with action sampling

The main drawback of OLT from the previous section is that each time it expands a leaf node, it has to compute *all* possible successor states (i.e., the outcome of each individual action is generated), which is clearly not feasible for large or continuous action spaces. We now present a simple extension of the OLT scheme where, instead of expanding all the actions, we only expand a small subset of them. This is done by defining a state-dependent parameterized probability distribution (or density) over the action space, which we will call the *action sampling heuristic*, and by optimizing its parameters with global optimization in the offline learning stage, jointly together with the parameters of the node scoring heuristic. Given the learned action sampling heuristic, every time a node is expanded, only a predefined finite number of actions drawn from this heuristic is expanded.

OLT with action sampling eliminates one of the main weaknesses of OLT: the fact that they deal with only finite and small action spaces. On the downside, it can however be more tricky to set up and shifts even more responsibility to the offline learning phase as it adds another set of variables to an already fairly high-dimensional global optimization problem.

Formally, we denote our action sampling heuristic by the induced distribution $p_{\text{act}}(a|\mathtt{n};\theta)$, which is the probability of selecting action $a \in A$ in leaf node $\mathtt{n} \in \mathcal{T}$ under learned parameters θ (analogously for densities). Let N_{act} denote the fixed number of samples drawn from p_{act}. The basic algorithms from Figure 3 are then simply modified by replacing the actions a_i by samples $\tilde{a}_i \sim p_{\text{act}}(\cdot\,|\mathtt{n}^*;\theta)$.

How to parameterize p_{act}? For the case of continuous action spaces one simple choice is to use a Gaussian with mean and variance being a parameterized function of the state: $p_{\text{act}}(\cdot\,|\mathtt{n};\theta)=\mathcal{N}\big(\cdot\,|\mu(\mathtt{n};\theta^1),\sigma^2(\mathtt{n};\theta^2)\big)$, where $\theta=(\theta^1,\theta^2)$ are the concatenated parameters. In our experiments, we will use second order monomials (for notational convenience here shown without removing double terms)

$$\mu(\mathtt{n};\theta^1)=\sum_{i=1}^{n_x}\theta_i^1\cdot\mathtt{n}.x_i+\sum_{i=1}^{n_x}\sum_{j=1}^{n_x}\theta^1_{j+i\cdot n_x}\cdot\mathtt{n}.x_i\cdot\mathtt{n}.x_j \quad (7)$$

$$\sigma^2(\mathtt{n};\theta^2)=\sum_{i=1}^{n_x}\theta_i^2\cdot\mathtt{n}.x_i+\sum_{i=1}^{n_x}\sum_{j=1}^{n_x}\theta^2_{j+i\cdot n_x}\cdot\mathtt{n}.x_i\cdot\mathtt{n}.x_j \quad (8)$$

in univariate action spaces. As before, other forms of parameterization for $\mu(\mathtt{n};\theta^1)$ and $\sigma^2(\mathtt{n};\theta^2)$, such as neural networks, would also be possible.

Remark: note that from a conceptual point of view OLT with action sampling now depends on three abstract functions: (1) the node scoring heuristic; (2) the action selection heuristic; and (3) the action sampling heuristic, each of which can be parameterized in various ways. Also note how the parameterization we use for the action sampling heuristic is more complex and contains higher order terms. This is so because, unlike the node scoring and action selection heuristics, which are both ranking functions, the action sampling heuristic must be capable of producing specific values (i.e, the means of the action distribution).

V. OLTs with recursive action splitting

We now introduce a second alternative extension of OLTs to deal with vector-valued action spaces $A \subset \mathbb{R}^{n_u}$ bounded by a box (i.e., A is a hyperrectangle). The core idea of this extension is to first transform the problem with continuous actions into a problem with discrete actions and then to use look-ahead trees for state-action pairs on top of the transformed problem. This idea is conceptually related to [11], who proposed a similar transformation of a continuous action MDP into a binary action augmented MDP in the context of value function based reinforcement learning.

A. Problem transformation

Given the initial problem (f,ϱ) with state space X and action space A, we define the new problem (f',ϱ') as follows:

- The new state space is the cartesian product of the initial state space and the set of action subspaces: $X'=X\times\mathcal{P}(A)$. Each state $x'=(x,\alpha)$ is composed of an original state $x\in X$ and an action subspace $\alpha\subset A$. Given root state x_0, the corresponding new state is $x'_0=(x_0,A)$.
- The new action space is composed of three discrete actions $A'=\{split\_left, split\_right, go\}$.
- The transition function f' is computed as follows:

$$\begin{aligned}f'((x,\alpha),split\_left)&=(x,left(\alpha))\\ f'((x,\alpha),split\_right)&=(x,right(\alpha))\\ f'((x,\alpha),go)&=(f(x,center(\alpha)),A)\ ,\end{aligned}$$

where $left(\alpha)$ and $right(\alpha)$ are two disjoint parts of the current action subspace, i.e., $left(\alpha)\cup right(\alpha)=\alpha$, and where $center(\alpha)$ is the central point of subspace α. In the case of one-dimensional action spaces $[l,u]$, we define $left([l,u])=[l,m]$, $right([l,u])=[m,u]$ and $center([l,u])=m$, with $m=\frac{l+u}{2}$. Note that although we only illustrate the one-dimensional case in the following, the ideas developed here can easily be extended to multi-dimensional action spaces (see [2]).

- The reward function ϱ' is computed as follows:

$$\varrho'((x,\alpha),a) = \begin{cases} \varrho(x, center(\alpha)) & \text{if } a = go \\ 0 & \text{otherwise.} \end{cases}$$

In order to illustrate our problem transformation, let us assume a system with a one-dimensional action space $A = [0,1]$, in which we want to apply action $a = 0.125$ to the current state x_0. This can be performed as follows. The initial transformed state is $x' = (x_0, [0,1])$. We start by applying the action $split\_left$, which leads to the new state $(x_0, [0, 0.5])$. We then apply the same action again, leading to $(x_0, [0, 0.25])$. Finally, we can now use the action go, which leads us to the new state $(f(x_0, 0.125), [0,1])$ and gives us a reward $\varrho(x_0, 0.125)$. Note that every time we execute the go action, the action subspace α is reset to cover the whole action space A.

B. OLTs on the transformed problem

Now that we have transformed our continuous action problem into a discrete action problem, we can apply a traditional OLT algorithm on top of the transformed problem. One key remark here is that only the go action involves running the simulator f and that applying the $split$ actions incurs nearly zero computational cost. In consequence, the online budget now corresponds to the number of times that a go action is executed, which is a reasonable simplification as long as the action space is not split excessively often. To avoid such an extreme situation, we allow split actions as long as the size of the region is above a certain minimum size Δ.

Since expanding a node with look-ahead trees over states involves simulating each of the available actions, each node expansions involves running the go action once, hence running f once. Since we want to be able to refine the action subspace several times before applying the go action, we move to look-ahead trees over state-actions. With this variant of LTs, the go action is only performed when a go node is expanded.

The only thing remaining is to explain how the two heuristics exp_score and act_score are implemented. As should be clear by now, both are abstract functions and can be instantiated in whatever way we like (and is reasonable). Here, we will only parameterize the node scoring heuristic and leave the action selection heuristic from Eq. (5) unchanged. To parameterize $\text{exp\_score}(\mathtt{n};\theta)$, we use a parameter vector $\theta = \left(\theta^{\text{left}}, \theta^{\text{right}}, \theta^{\text{go}}\right)$ consisting of the three blocks $\theta^{\text{left}}, \theta^{\text{right}}, \theta^{\text{go}} \in \mathbb{R}^{5n_x}$, where each block gets activated only for the corresponding discrete action stored in $\mathtt{n}.a \in \{\text{left,right,go}\}$. The parameterization we use is thus

$$\text{exp\_score}(\mathtt{n};\theta) = \sum_{i=1}^{n_x} \mathtt{n}.x_i\big(\theta_i^{\mathtt{n}.a} + \mathtt{n}.r\theta_{i+n_x}^{\mathtt{n}.a} + \mathtt{n}.d\theta_{i+2n_x}^{\mathtt{n}.a} + nsplit(\mathtt{n}.\alpha)\theta_{i+3n_x}^{\mathtt{n}.a} + center(\mathtt{n}.\alpha)\theta_{i+4n_x}^{\mathtt{n}.a}\big), \quad (9)$$

where $(x,\alpha) = \mathtt{n}.x$ and where $nsplit(\mathtt{n}.\alpha)$ denotes the number of splits that have been applied to obtain α from A.

VI. Simulation Experiment

The goal of the following simulation experiment is to give the reader a general idea of what kind of results can be achieved with the OLT framework and how the various building blocks from the previous sections can be put together. Because of the large number of possible combinations, we will only give a very limited demonstration of what is possible.

A. The benchmark domains

We carry out our experiments in the following domains:

Inverted pendulum: The inverted pendulum swing-up task is a nonlinear control problem with 2-dimensional state and 1-dimensional action space (the problem is still small enough to solve the HJB equation on a high-resolution grid). The objective of the task is to swing up and stabilize a single-link inverted pendulum. The dynamics of the system f, the reward function ϱ, and physical parameters we used to instantiate this problem are exactly as in [7] (Appendix A). Note that, unlike [7], where the action space was discretized to the 5 choices $a \in \{-5, -2.5, 0, +2.5, +5\}$, we will here use the full continuous action space $A = [-5, 5]$.

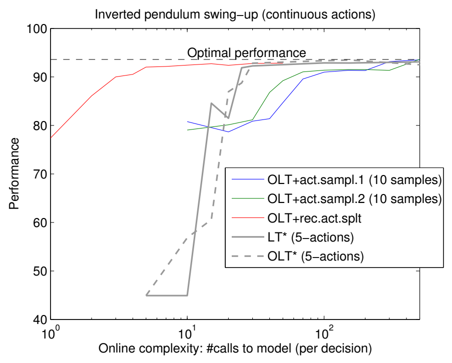
HIV drug treatment: The HIV drug treatment problem is a nonlinear control problem with 6-dimensional state and 2-dimensional action space. The objective of the task is to optimize the treatment of a patient infected by HIV over a period of a few years. The treatment of the patient consists of choosing a combination of two drugs to administer every 5 days. Administering the correct cocktail with the correct timing can hinder the spread of HIV-infected cells and will eventually bring the patient into a healthy state (a locally stable equilibrium); however, the drugs also have side effects on the patient's health and thus their use should be kept to a minimum. The dynamics of the system f, the reward function ϱ, and physical parameters are exactly as in [7] (Appendix D). Note that, unlike [7], where the action space was discretized to the 4 on/off choices $a \in \{(0,0), (0,0.3), (0.7,0), (0.7,0.3)\}$, we will here use the full continuous action space $A = [0, 0.7] \times [0, 0.3]$.

B. Experimental protocol

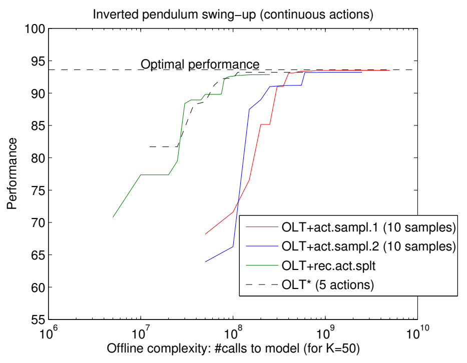
We examine the five methods shown in Figure 4. Note that OLT and LT are only included as baseline methods: the results are directly copied from [7] and were obtained for the discretized action space, hence these methods do not perform any form of action sampling. The implementation details in Figure 4 are largely self-explanatory. For the offline optimization of the parameters θ we used a simple estimation of distribution algorithm (EDA) as in [7] with 200 samples for the inverted pendulum and 800 samples for the HIV drug domain, 50 iterations, and recomputing the parameters of the sampling distribution over the best 10%. (Our interest here was not to be maximally efficient in our offline optimization, so these non-tuned settings were just fine.) Note that each

Algorithm	Nodes	Node scoring	Action scoring	Action sampling	Online complexity	Num. params	Implementation details
OLT+act.sampl.1	states	generic	generic	optimized (explicit)	#node expansions × #samples	10/117	Global search solver: EDA. Action sampling: Eqs. (7),(8). Node scoring: Eq. (4). Action selection: Eq. (5).
OLT+act.sampl.2	states	optimized	generic	optimized (explicit)	#node expansions × #samples	16/129	Global search solver: EDA. Action sampling: Eqs. (7),(8). Node scoring: Eq. (6). Action selection: Eq. (5).
OLT+rec.act.splt.	state-act	optimized	generic	optimized (implicit)	#node expansions	30/90	Global search solver: EDA. Node scoring: Eq. (9). Action selection: Eq. (5).
OLT as in [7]	states	optimized	generic	none	#node expansions × #actions	6/18	Global search solver: GPO. Node scoring: Eq. (6). Action selection: Eq. (5).
LT as in [6]	states	generic	generic	none	#node expansions × #actions	0	Node scoring: Eq. (4). Action selection: Eq. (5).

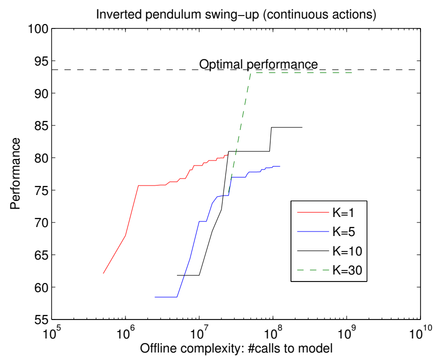
Fig. 4. The variants of OLT/LT we consider in our experiment at a glance. Entries in the column "num. params" give the dimensionality of θ, i.e., the dimensionality of the search space for the inverted pendulum domain (2-dim state space) and the HIV drug domain (6-dim state space).



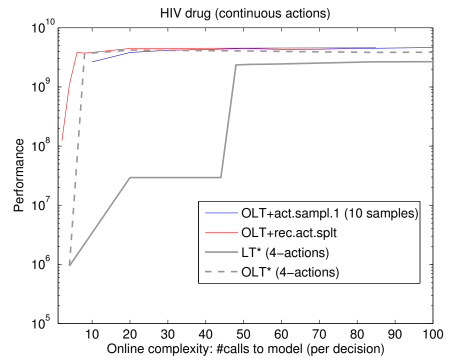
(a) Performance vs online complexity



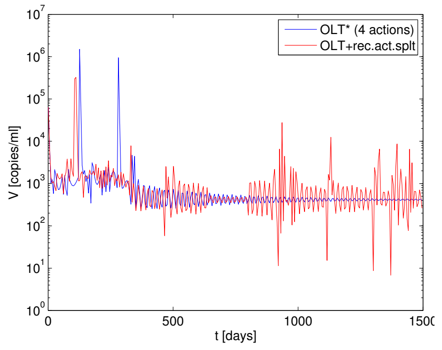
(b) Performance vs offline complexity ($K = 50$)



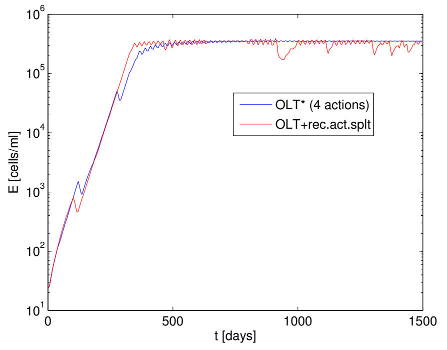
(c) Learning curve OLT.act.smpl.1



(d) Performance vs online complexity



(e) Trajectory $x_5(t)$ (infected cells)



(f) Trajectory $x_6(t)$ (healthy cells)

Fig. 5. Results for the inverted pendulum domain (top row) and the HIV drug domain (bottom row). The curves for LT* and OLT* are only shown as a reference; they were obtained for discretized action spaces and are directly copied from [7].

parameter setting θ must be specifically optimized for one particular setting of the online budget K. To evaluate the performance of each method, we run the resulting policy $\pi_{f,\varrho}(\cdot\,;\theta)$ for $H = 500$ steps in the inverted pendulum and $H = 300$ steps in the HIV drug treatment domain and record the discounted cumulative reward (see Eq. (2)) attained in each case.

C. Results

Figure 5 compares the result we obtain over all methods for different settings of the online budget K:

Inverted pendulum: Figure 5a plots performance vs. online complexity (after completing 50 EDA iterations in the offline optimization of the parameters) and shows that generally all the methods are able to attain a very close to optimal performance level; OLT+rec.act.splt is, however, particularly efficient and realizes a substantially better cost-benefit ratio. Figure 5b plots performance vs. offline complexity (for a fixed online budget of $K = 50$) and confirms this picture; here performance means the best performance among all parameter settings θ evaluated by EDA so far, and offline complexity means total number of calls to the model (which equals the online complexity multiplied by the episode length H multiplied by the current number of θs evaluated). Finally, Figure 5c shows the learning curves for the method OLT+act.sampl.1

over various online budget settings K; i.e., it plots how the performance improves in relation to the offline complexity invested for the given constraint K (and depending on the number of θs evaluated so far by EDA).

HIV drug treatment: Similarly, Figure 5d plots performance vs. online complexity in the HIV domain. The results show that the two new methods, OLT+act.sampl.1 and OLT+rec.act.splt, are able to achieve a better performance in the continuous action formulation of the domain than the previously best known method OLT was able to achieve in discrete action formulation: namely $4.78e9$ vs $4.21e9$. We also observe that, again, OLT+rec.act.splt is able to realize a good cost-benefit ratio. Figure 5e-f examines in more detail the induced policies and compares the controlled trajectory for the 5th state variable (which measures the number of infected particles and enters negatively into the per-step reward calculation) and the 6th state variable (which measures the number of healthy cells and enters positively into the per-step reward calculation). It can be seen that OLT+rec.act.splt produces qualitatively a different behavior than OLT: while there are fewer large spikes of infected particles at the beginning, the behavior at the end is noticable less smooth. It should be noted though that the discount factor γ was set to $\gamma = 0.98$ as in all previous work, and thus puts less emphasis on the behavior at later stages of the episode (as long as it stays in the healthy equilibrium).

VII. Related work

Continuous action selection in RL: In the past, many approaches have been proposed to deal with continuous actions in RL. Since most of them are applied in the context of value function based RL, they are typically not directly compatible with LT/OLT type policies. One relevant work is [11], which is comparable to OLT+rec.act.splt in that both turn the continuous action problem into an augmented binary action problem, where the original state space is augmented to include the current partition of the action space, and the binary action just refines and narrows down this partition to a single value (in [11] until a minimum resolution is reached, whereas here we include a "go" action to interrupt this recursive refinement).

Look-ahead tree policies: OLT was first introduced in [9], whereas what we understand by LT was introduced in [6] and is also studied in [3]. In general, look-ahead tree based policies are of course widely used in planning and search, and we can only sketch the most important relations here: LT/OLT is related to model predictive control [4], [10] in that both are model-based and consider all possible future developments of the system over a short time window in a receding horizon manner. MPC techniques can perform exceedingly well; however, they rely on strong regularity assumptions on the system dynamics and the reward function (e.g., linearity) to reformulate the search for an optimal open-loop sequence of actions as a standard mathematical programming problem and thus are not everywhere applicable. LT/OLT is also related to standard planning and heuristic search methods such as A^* [5], and extensions aimed at learning the heuristics driving the expansion of the tree [1]. Finally, LT/OLT is also related to Monte-Carlo tree search (MCTS) [8], with the tree-development strategy of OLT being the equivalent of the so-called *selection phase* of MCTS. One key difference with MCTS is that OLT does not rely on random simulations (until termination) to estimate state values.

VIII. Conclusion

This paper studies look-ahead tree based control policies from the viewpoint of computational costs and makes explicit the constraint that in real world applications only limited computational resources are available. We describe two extensions to the basic OLT family of algorithms that allow tackling optimal control problems with large and continuous action spaces specifically under the constraint of a fixed online budget (expressed in terms of allowed number of calls to the generative model): one that is based on action-sampling and one that is based on recursive action splitting. The underlying idea and improvement over our previous work on OLT is to parameterize not only which nodes are expanded but also which actions to consider for expanding, effectively implementing a look-ahead tree over nodes which are state-action pairs instead of just states. Our (thus far quite limited) experimental evaluation confirms that generally both extensions allow OLT to scale to work also for continuous action spaces and also demonstrates that additional savings in the online complexity can be realized.

References

[1] S.J. Arfaee, S. Zilles, and Holte R.C. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175:2075–2098, 2011.

[2] S. Bubeck, R. Munos, G. Stoltz, and C. Szepesvári. *X*-armed bandits. *Journal of Machine Learning Research*, 12:1655–1695, 2011.

[3] L. Busoniu, R. Munos, B. De Schutter, and R. Babuska. Optimistic planning for sparsely stochastic systems. In *Proc. of IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL-11)*, pages 48–55, 2011.

[4] D. Ernst, M. Glavic, F. Capitanescu, and L. Wehenkel. Reinforcement learning versus model predictive control: a comparison on a power system problem. *IEEE Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics*, 39(2):517–529, 2009.

[5] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[6] J-F. Hren and R. Munos. Optimistic planning of deterministic systems. In *Proc. of European Workshop on Reinforcement Learning*, pages 151–164. Springer, 2008.

[7] T. Jung, L. Wehenkel, D. Ernst, and F. Maes. Optimized look-ahead tree policies: A bridge between look-ahead tree policies and direct policy search. *International Journal of Signal Processing and Adaptive Control*, 2013 (to appear).

[8] L. Kocsis and C. Szepesvári. Bandit based Monte Carlo planning. *Proceedings of the 17th European Conference on Machine Learning (ECML 2006)*, pages 282–293, 2006.

[9] F. Maes, L. Wehenkel, and D. Ernst. Optimized look-ahead tree policies. In *Proceedings of the 9th European Workshop on Reinforcement Learning (EWRL 2011)*, 2011.

[10] M. Morari and J.H. Lee. Model predictive control: Past, present and future. *Computers & Chemical Engineering*, 23(4):667–682, 1999.

[11] J. Pazis and M. Lagoudakis. Binary action search for learning continuous-action control policies. In *Proc. of ICML*, 2009.