Boundary Extension Features for Width-Based Planning with Simulators on Continuous-State Domains

Florent Teichteil-Königsbuch[1] , **Miquel Ramirez** [2] and **Nir Lipovetzky** [3]
[1]Airbus, Artificial Intelligence Research
[2]Dept. Electrical and Electronic Engineering, The University of Melbourne
[3]School of Computing and Information Sciences, The University of Melbourne
florent.teichteil-koenigsbuch@airbus.com,{miquel.ramirez,nir.lipovetzky}@unimelb.edu.au

Abstract

Width-based planning algorithms have been shown to be competitive with state-of-the-art heuristic search and SAT-based approaches, without requiring access to a model of action effects and preconditions, just access to a black-box simulator. Width-based planners search is guided by a measure of the novelty of states, that requires observations on simulator states to be given as a set of features. This paper proposes agnostic feature mapping mechanisms that define the features online, as exploration progresses and the domain of continuous state variables is revealed. We demonstrate the effectiveness of these features on the OpenAI gym "classical control" suite of benchmarks. We compare our online planners with state-of-the-art deep reinforcement learning algorithms, and show that width-based planners using our features find policies of the same quality with significantly less computational resources.

1 Introduction

Width-based search methods have been used successfully with game simulators [Lipovetzky *et al.*, 2015; Geffner and Geffner, 2015; Shleyfman *et al.*, 2016; Jinnai and Fukunaga, 2017], as well as, classical [Lipovetzky and Geffner, 2012; Lipovetzky and Geffner, 2017a], non-deterministic [O'Toole *et al.*, 2019], and multi-agent [Gerevini *et al.*, 2019] planning. All these works have successfully addressed problems with discrete state variables, yet we argue that the most natural setting for simulation-based planning is that with continuous state variables, which has been seldom worked on [Ramirez *et al.*, 2018]. We also perceive a vast untapped potential for width-based methods when it comes to address control problems over complex non–linear systems, such as those typically modeled and portrayed in aircraft and spacecraft simulators [Stevens *et al.*, 2015].

Continuous state simulations pose major challenges for existing width-based search algorithms to be applicable. Handling efficiently continuous state variables is an open problem, since naively mapping them onto finite-domain variables results in search trees of colossal size. Multi-dimensional control inputs, or continuous actions need to be discretised, resulting in huge branching factors. To both of these problems we offer an answer in the form of a dynamic, adaptive feature map algorithm, the *boundary extension encoding* (BEE), that handles efficiently continuous variables and generates positive synergies with existing search strategies to deal with the blow up in branching factors.

The paper is structured as follows. Section 2 includes essential background on optimal control problems over general dynamical systems, classical planning over simulators and width-based search. Section 3 describes a novel adaptive algorithm for defining feature maps to calculate the novelty of states. In Section 4 we evaluate experimentally the dynamic feature maps over OpenAI's `gym` suite of classical control problems, and we compare against state-of-the-art deep reinforcement learning (DRL) and optimal control techniques. We conclude discussing related and future lines of work.

2 Background

2.1 Optimal Control Problems over Simulators

We start by describing the systems of interest in a very generic way due to [Willems, 2007], as we do not make any special assumptions on the form of dynamic and state constraints. We consider the space of observable signals to be an arbitrary n dimensional subset of $\mathbb{R}$, $\mathcal{X} \subset \mathbb{R}^n$, with a given initial configuration $q_0 \in \mathcal{X}$. The set of feasible trajectories over $\mathcal{X}$ is modeled as a dynamical system $\Sigma = (\mathbb{T}, \mathcal{X}, \mathcal{B})$. $\mathbb{T}$ is the interval $[0, \mathcal{T}]$ where $\mathcal{T}$ is a positive real number. $\mathcal{B}$ is the *behaviour* of the model, a subset of $\mathcal{X}^{\mathbb{T}}$, the set of all maps from $\mathbb{T}$ to $\mathcal{X}$, and formalizes what trajectories or *paths* $\tau : \mathbb{T} \to \mathcal{X}$ are possible. Each $q(t) \in \mathcal{X}$ in paths $\tau \in \mathcal{B}$ needs to satisfy a given set of differential path constraints describing what changes on signals q are possible:

$$\forall t \in \mathbb{T} : h(q(t), \dot{q}(t), \ddot{q}(t)) = 0, \quad (1)$$

$$g(q(t), \dot{q}(t), \ddot{q}(t)) \leq 0, \quad (2)$$

with the additional constraint that $q(0) = q_0$. Of interest to us are control problems where it is required that trajectories end on a specific target or goal point in configuration space, q^*, so we will further require that $q(\mathcal{T}) = q^*$.

In this paper, we assume that we do not have available a description of constraints h and g, but rather we have access to a *simulator*, a procedure which is guaranteed to generate only trajectories $\tau \in \mathcal{B}$. The simulator discretises $\mathbb{T}$, that becomes

a *finite* subset of $\mathbb{R}^{0+}$, $\mathbb{T} = \{0, \Delta t, 2\Delta t, \cdots, T\Delta t\}$ where Δt is a positive real number and $T\Delta t = \mathcal{T}$. For brevity, we refer to each $t_k \in \mathbb{T}$ by their index $k \in \{0, 1, \cdots, T\}$ and similarly we shorthand $q_k \coloneqq q(t_k)$. At every time step k, or *control cycle*, the simulator requires an input $u(t_k) \in U \subset \mathbb{N}$, abbreviated as u_k, in order to *steer* the evolution over time of the dynamical system Σ. We loosely define the simulator as a function $F : \mathcal{X} \times U \to \mathcal{X}$. F can be defined in many ways, but we will assume that inputs map some or all of $\dot{q}(t)$ or $\ddot{q}(t)$ to a function over $\mathcal{X}$ or constant real values, and q_{k+1} is calculated with a numerical integration algorithm.

An optimal control problem [Bertsekas, 2017] over a simulator F is then defined as the problem of seeking sequences of control inputs that maximize a given performance index, which we formalise as follows

$$\max_{u_0,\ldots,u_{T-1}} \ell_T(q_T) + \sum_k \ell(q_k, u_k) \tag{3}$$

$$s.t.\ \ q_{k+1} = F(q_k, u_k) \tag{4}$$

$$q_T = q^* \tag{5}$$

ℓ_T and ℓ are respectively the *terminal* and *stage* cost functions. When $\ell_T(q_T) = 0$, the problem above becomes a classical planning problem [Bonet and Geffner, 2013] over a finite horizon T, since U is discrete. If equation (5) is dropped, it becomes a *net-benefit* planning problem [Helmert *et al.*, 2008; Keyder and Geffner, 2009].

2.2 Factored State Models and Simulators

Francès et al [2017] have noted that purely declarative planning languages are not the only way to represent classical planning models compactly, as illustrated by the discussion above. Indeed, state models $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c\rangle$ can also be represented in general by using state variables and by encoding the functions $A(s)$, $f(a, s)$, and $c(a, s)$ as *black box*[1] procedures.

For this, in order to represent the dynamic system $\Sigma = (\mathbb{T}, \mathcal{X}, \mathcal{B})$, we adapt the definition by Francès et al [2017] of a *factored state model* as a tuple $\mathcal{F} = \langle \mathcal{X}, D, s_0, G, Act, A, f, c\rangle$. $\mathcal{X}$ is the set of variables, their possible values given by D, the domain D_x of variable $x \in \mathcal{X}$. s_0 is the initial assignment to the variables compatible with their domains, G is a set (conjunction) of goal Boolean conditions over terminal states q_T expressing Equation 5. Act is a one-to-one surjective map onto U. Lets denote S as the set of possible assignments (states) $s : \mathcal{X} \to D$, then $A : S \mapsto 2^{Act}$, $f : S \times A \mapsto S = F(q, u)$, and $c : S \times A \to \mathbb{R} = -\ell(q, u)$ are all functions given as *procedures* that compute the set of actions applicable in each state, successor states, and the cost incurred by doing an action. The tuple $\mathcal{F}$ provides a compact representation of the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c\rangle$ where S is the set of states as defined above, and S_G is the set of assignments satisfying each goal condition in G. While in planning it is common for costs to be positive, as noted above, we consider the net-benefit setting, where instead of costs we have rewards.

[1] "Black box" meaning that the planner can evaluate these functions, but does not have access to their internal structure or a symbolic representation.

2.3 Width-Based Search

The content of this Section summarises the state-of-the-art in width-based search as discussed elsewhere [Lipovetzky and Geffner, 2012; Lipovetzky and Geffner, 2017a; Bandres *et al.*, 2018]. We refer the reader to these papers for a more thorough discussion of the algorithms and concepts below.

Width-based search is a blanket term referring to a diverse set of classic blind and heuristic search algorithms, adapted to use the so–called *novelty heuristics* first proposed by Lipovetzky and Geffner [2012]. In general terms, these heuristics proceed to determine whether a state s, with generation order $e(s)$[2] contains values of variables not present in any other state s' with generation order $e(s') < e(s)$. Based on this simple and effective idea, follow-up works have proposed several methods to define the *novelty measure*, $w(s)$, along with search algorithms designed to exploit them and seek synergies with other heuristics. We discuss next two recent approaches that have been demonstrated to be state-of-the-art in offline and online planning.

k-BFWS(f) [Lipovetzky and Geffner, 2017b] is an incomplete, polynomial-time, greedy best-first search (GBFS) algorithm where the evaluation function f is such that nodes are ordered by their novelty and an estimate of costs-to-go. Nodes are pruned if their novelty $w(s)$ is greater than a given bound k. The novelty heuristic $w(s)$ is defined over a set of functions[3] that partition S, so only states s' in the same partition than s are considered to determine if novel information is being contributed by s.

Rollout IW (RIW) [Bandres *et al.*, 2018] is a rollout algorithm like MCTS [Bertsekas, 2017] that constructs a depth-first lookahead over a fixed horizon T, designed around a definition of $w(s)$ substantially different from previous approaches. Bandres et al. introduce a set of *dynamic* features $\phi_{x,v} : S \to [1, T]$ for each state variable $x \in \mathcal{X}$ and value $v \in D_x$, initially defined as $\phi_{x,v}(s) := T$. Whenever a state s_t, with time index or depth t is generated, it is determined if exists a feature $\phi_{x,v}(s) > t$ where $x_v \in s_t$. If such a feature exists, then $w(s_t) = 1$, and $\phi_{x,v}(s) := t$. Otherwise, $w(s_t)$ is set to an arbitrary constant other than 1, the node in the lookahead is marked, and a label is propagated backwards to its parent. Marked nodes are not expanded. RIW terminates whenever the label is backpropagated all the way to the initial state.

3 Features for Measuring Novelty

Previous research on width-based algorithms for planning has focused on problems with discrete state variables, where states are given as sets of atoms, and features map sets of atoms to Boolean values or some subset of $\mathbb{N}$. A notable exception to this is the work by Ramirez et al. [2018], where state variables are real-valued so $D_x \subset \mathbb{R}$. We start noting that in models of computation that support only *finite* precision arithmetic, it holds that the number of states generated during search is finite, as long as a solution can be found in

[2] Initially, $e(s_0) = 0$. For the first successor s' of s_0 generated by a given algorithm, $e(s') = 1$ and so on.

[3] Lipovetzky and Geffner use mainly heuristic cost-to-go approximations. Other functions can be used too.

a finite number of steps. This is not in contradiction with the undecidability results by Helmert [2002], as these are predicated on infinite arithmetic.

Ramirez et al. define features $\phi_x : S \to [0, 2^B - 1]$ mapping states to a subset of $\mathbb{N}$, where B is the number of bits in the floating–point representation of variable values. ϕ_x can be evaluated in constant time, as long as the programming language used allows to access the *datum* [Stepanov and McJones, 2009] or binary encoding, for floating-point datatypes. While this may seem naive, Ramirez et al. [2018] proved it to be very effective. In *goal–based* tasks, where no positive reward is attained until the goal is reached, or when rewards are sparse, it suffers from significant issues that negate the advantage of systematic yet efficient exploration associated with width-based search methods. These limitations can be illustrated by the well–known benchmark for Reinforcement Learning, *Mountain Car*, due to [Moore, 1990; Boyan and Moore, 1995]. States in *Mountain Car* consist of two variables, *position* p and *velocity* v, with three actions, *left*, *right* and *do-nothing*. The former two actions respectively set $\dot{v}$ to a negative (positive) constant, and then integrate the dynamics of the system for a given duration. The task requires the car to reach a goal position p^*, and the initial state is set to the vector $[0\,0]^T$ where the car is at rest and has zero potential energy.

Mountain Car defeats Ramirez et al. approach as follows. For a given search depth d, only a small subset of plans out of 3^d possible plans *consistently* increase potential energy by moving the car away from the initial state. As the car gains potential energy proportional to d, this allows it to attain higher velocities when the *do-nothing* action is applied. This sets the car into the right track to eventually reach p^*, as the simulator is designed in such a way that the car needs to swing back and forth a number of times before reaching the goal. Most of the 3^d paths are unrelated to optimal plans. Under these conditions, Ramirez et al. features do not discriminate much, and the lookahead size grows so large that runtimes to determine the best action is orders of magnitude larger than the duration set for the control cycle.

3.1 Boundary Extension Encoding – BEE

The previous discussion motivates our framework, the *Boundary Extension Encoding*, or BEE for short, to define features for width-based planning algorithms over continuous state variables. We now summarise the key intuitions. First, we propose features that go beyond static discretisations, as they take into account the dynamics of the problem and choices made by the search algorithm. Second, we have designed these features to produce novelty measures of states such that a state is considered "novel", iff it "pushes the envelope", as defined by the valuations already encountered during the search. Doing so we overcome the challenge posed by states that are not part of any optimal trajectory, yet differ only by small amounts from previously generated trajectories.

Figure 1 illustrates the operation of the BEE encoding, defining features on–line, as the search progresses. We denote as $x_s \in D_x$ the value of a state variable $x \in \mathcal{X}$ in state s. For each variable $x \in \mathcal{X}$, BEE keeps track of the new boundary extensions of states visited during the search. Initially, the ordered list of boundary extensions associated with variable x, $B^+(x)$ for positive extensions and $B^-(x)$ for negative extensions, consists of a single element, $B^+(x) = B^-(x) = (x_{s_0})$ where x_{s_0} is the value of state variable x in the initial state s_0. $B^+(x)$ is sorted in increasing order, while $B^-(x)$ is sorted in decreasing order, making sure x_{s_0} is always the first element in the lists. Every time a new state s is generated where x_s is greater than the positive boundary pb_x, i.e. the maximum element in $B^+(x)$, a new element x_s is added into $B^+(x)$ extending its positive boundary. Analogously, whenever x_s is less than the lowest negative boundary nb_x in $B^-(x)$, a new boundary x_s is recorded, extending the negative boundary nb_x. The BEE features evaluated on s, $\mathcal{B}(s)$, is the set of boundary indexes $\{\beta_x = i \in \mathbb{Z} \mid x \in \mathcal{X},\, x_s > x_{s_0}\}$ and $\{\beta_x = -i \in \mathbb{Z} \mid x \in \mathcal{X},\, x_s < x_{s_0}\}$ such that x_s is contained in the interval $(x_{i-1},\, x_i]$ where $x_{i-1},\, x_i \in B^+(x)$ or the interval $[x_i,\, x_{i-1})$ where $x_i,\, x_{i-1} \in B^-(x)$. Namely, β_x represents the index in the ordered list containing the ith boundary extension that is greater than x_s if $x_s > x_{s_0}$, or the ith boundary extension that is smaller than x_s. Indices in the negative boundary extensions are represented as negative indices to differentiate them from their positive counterparts. The greatest positive boundary pb_x and smallest negative boundary nb_x outline the *explored boundary* of D_x.

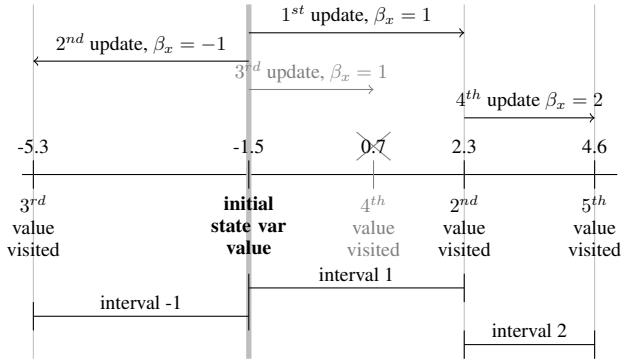

Figure 1: Example of the boundary extension encoding (BEE) on a single state variable. The value 0.7 has $\mathcal{B} = \{\beta_x = 1\}$ because it does not contribute to extend the boundaries of the visited values of the state variable (it falls in the 1^{st} visited interval). $w(s)$ will be greater than 1 and s will be pruned if no other boundary is extended.

When a state s is generated such that the value x_s for variable x falls between $pb_x \geq x_s > x_{s_0}$ or $nb_x \leq x_s < x_{s_0}$, it follows that it does not extend the boundary of the *explored* domain of x. In Figure 1 we can see how the value 0.7 belongs to the interval $(-1.5, 2.3]$, which was previously generated when the explored boundary of x was extended when we came across value 2.3. As the interval represented by the feature $\beta_x = 1$ has already been observed, variable x alone would not contribute to change the novelty measure of s.

The BEE features may look to be quite specific to domains like *Mountain Car*, where optimal plans extend as much as possible the explored boundary of the domains of certain variables. Yet our experiments show that they are useful in several other continuous-state control domains, as well. Somewhat surprisingly, the BEE features turn out to be quite useful

as well on net-benefit problems such as the classic *Cart Pole* problem in Reinforcement Learning [Sutton, 1996]. This observation is explained by pointing out that the BEE features dynamically create an abstraction of $\mathcal{X}$, that promotes exploration and results in a compact search tree. Width-based algorithms then reach quickly the horizon T in Eqs. (3)–(5) and back propagate information about dead-end states to avoid, in *Cart Pole*, the pole touching the ground, or states rated poorly by the cost function in Eq. (3).

3.2 BEE Features and Measuring Novelty

We formalize next the notions of boundaries, features and novelty over the set of BEE features.

Definition 3.1 (BEE Boundaries)**.** *Given a* factored state model $\mathcal{F}$*, the ordered list of BEE positive* B^+ *and negative* B^- *Boundaries is defined as* $B^+(x) = (\ x_0, \ldots, x_n \mid x_i \in D_x,\ x_i < x_j,\ i < j\}$ *and* $B^-(x) = (\ x_0, \ldots, x_n \mid x_i \in D_x,\ x_i > x_j,\ i < j\}$ *for every variable* $x \in \mathcal{X}$*.*

Boundary lists are sorted and contain the boundary extensions that define contiguous intervals over each state variable. Let $nb_x = \min B^-(x)$, and $pb_x = \max B^+(x)$.

Definition 3.2 (BEE Boundary Updates)**.** *Given a state sequence* $s_0, \ldots, s_i, \ldots, s_n$*, the boundaries* $B^+(x) = B^+_{s_i}(x)$ *and* $B^-(x) = B^-_{s_i}(x)$*, for* $x \in \mathcal{X}$ *are updated as follows: (i)* $B^-_{s_0}(x) = B^+_{s_0}(x) = (x_{s_0} \mid x_{s_0} \in D_x)$*, (ii)* $B^+_{s_{i+1}}(x) = B^+_{s_i}(x) \bigcup N^+(x_{s_{i+1}})$*, (iii)* $B^-_{s_{i+1}}(x) = B^-_{s_i}(x) \bigcup N^-(x_{s_{i+1}})$*, where* $N^+(x_v)$ *and* $N^-(x_v)$ *are the sets of new intervals for value variable* x_v*:*

$$N^+(x_v) = \begin{cases} (x_v) & x_v > pb_x \\ \emptyset & \textit{otherwise} \end{cases} N^-(x_v) = \begin{cases} (x_v) & x_v < nb_x \\ \emptyset & \textit{otherwise} \end{cases}$$

A state can add at most $|\mathcal{X}|$ new boundaries as defined above. Let $B^+_s(x)$ and $B^-_s(x)$ be the list of boundaries created up to state s for each variable x.

Definition 3.3 (BEE Features)**.** *Given a state* s*, the set of active BEE features is defined as* $\mathcal{B}(s) = \mathcal{B}^+(s) \cup \mathcal{B}^-(s)$ *where* $\mathcal{B}^+(s) = \{\ \beta_x = i \mid x_s > x_{s_0},\ x \in \mathcal{X}\}$ *and* $x_s \in (x_{i-1}, x_i]$ *belongs to an interval defined by the ordered list of positive boundary extensions* $x_{i-1}, x_i \in B^+_s(x)$*. When* $x_s \in (x_i, x_{i-1}]$ *belongs to an interval within the negative ordered list of boundaries* $x_i, x_{i-1} \in B^-_s(x)$ *then* $\mathcal{B}^-(s) = \{\ \beta_x = -i \mid x_s < x_{s_0},\ x \in \mathcal{X}\}$*.*

The set $\mathcal{B}(s)$ contains the features for active intervals as indexes representing the intervals containing the value $x_s \in D_x$ for all state variables $x \in \mathcal{X}$ of the problem.

Definition 3.4 (Novelty over BEE features)**.** *Given a state sequence* $s_0, \ldots, s_{n-1}$ *of previously generated states, the novelty* $w(s_n)$ *of a new state* s_n *is the size* $|t|$ *of the smallest tuple* $t \in \mathcal{B}(s_n)$ *s.t.* $t \notin \mathcal{B}(s_i)$*,* $i = 0, \ldots, n-1$*.*

If we restrict ourselves to tuples of maximal size 1, the above simplifies to check that $\exists_{x \in \mathcal{X}} N^+(x_s) \cup N^-(x_s) \neq \emptyset$ holds. We note that when considering tuples of size 2, Definition 3.4 is sensitive to existing, yet unknown, couplings between pairs of variables. The use of BEE features for novelty over RIW and k-BFWS follows the same procedure described in Section 2.3, but uses the feature valuation $\mathcal{B}(s)$ instead of $\phi_{x,v}$.

4 Experiments

We run our experiments on an Amazon Web Service *p2.xlarge* instance with 4 Intel Xeon 2.30GHz processors, 61 GB of RAM and 1 NVIDIA K80 GPU. We compare k-BFWS with different evaluation functions, RIW, LQR and PPO2[4] on the set of OpenAI gym's classic control problems[5]. LQR is a Linear Quadratic Regulator controller [Perez *et al.*, 2012] while PPO2 is a widely used deep policy-gradient reinforcement learning algorithm [Schulman *et al.*, 2017]. All algorithms use a single CPU except PPO2 that makes use of 4 CPUs and 1 GPU. In all the presented experiments, k-BFWS and RIW use the BEE features encoding, and start with increasing values of novelty i from 1 until the problem is solved. All the tested benchmarks required to reach a width $i = 2$ except the *Cart Pole* benchmark which required $i = 3$ or $i = 4$ depending on the initial state. The fact that $i > 1$ is required shows that none of those benchmarks can be solved by trying to extend the boundaries of a single state variable independently from the others. k-BFWS stops searching as soon as it finds a solution, and RIW stops whenever the computation budget, measured in number of calls to the simulator, is consumed. We tried also naive features, $\phi_x : S \to D_x$ so a variable value is a feature, and Ramirez et al. features from binary representations described in Section 3. We do not report results for them since neither k-BFWS nor RIW could solve a single problem in the set of benchmarks considered.

Since the initial state is set randomly by simulators when restarting, we report performance metrics for k-BFWS and RIW averaged over several independent runs.

We implemented k-BFWS, RIW and the BEE encoding in the scikit-decide library [AIRBUS - AI Research, 2020].

Feature Composition and Base Policy for RIW

Bandres et al. present RIW using as the *base policy* a simple random walk. Instead, we use an ϵ-$novelty$ policy, that uses a one-step lookahead choosing the action that leads to the successor state that is not novel with probability $\propto \epsilon, 0 < \epsilon < 1$, and novel ones with probability $\propto \frac{1-\epsilon}{w(s)}$, thus favoring states with better (smaller) novelty. As noted in Section 3, we have a BEE feature β_x for every state variable x, so we have to redefine Bandres' features over these, rather than state variables. This is straightforward, as we map states s into $\beta_s = \{\beta_x(s) \mid x \in \mathcal{X}\}$, and implementing $\phi_{\beta_x,v}$ does not require to know beforehand the domain of x.

4.1 Goal-Based Problems

We discuss first *Mountain Car* and *Acrobot*, benchmarks where the task is to reach a goal state from a set of initial states, of which the simulator chooses one randomly. Both tasks are challenging because the underlying dynamics can make the agent perform very small steps that do not make much progress towards the goal state.

Figure 2 shows how PPO2's reward evolves during learning on *Mountain Car* with continuous actions. We also plot the results of k-BFWS with BEE as if it was a learning algorithm to ease comparison with PPO2: in fact, k-BFWS ini-

[4]PPO2 from https://github.com/hill-a/stable-baselines

[5]https://github.com/openai/gym/tree/master/gym/envs/classic_control

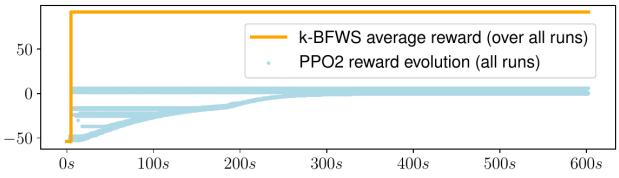

Figure 2: PPO2's reward evolution compared with k-BFWS-BEE on Mountain Car (100 runs).

tially spends time to compute a solution plan, represented by a vertical line on Figure 2, that does not change throughout its execution, so the mean reward for k-BFWS in Figure 2 flatlines. k-BFWS does not support continuous actions, so we use for it the discrete action formulation of *Mountain Car*. The evaluation function used by k-BFWS is $f = \langle w \rangle$, so the open list is ordered in ascending order of novelty measures.

The explanation for the simplistic f above being so effective on this problem is related to the reward function in this benchmark, that assigns 0 to every transition but the ones reaching a goal state, that attain a reward of 100. We could see, tracing the planner, that it is necessary to extend the boundaries $B^+(x)$ and $B^-(x)$ as quickly as possible, in order to have a chance for some state s_k, $k < T$, be a goal state, and hence, inform the lookahead with positive reward. Figure 2 shows that k-BFWS can solve *Mountain Car* in 4.6s on average. In contrast, PPO2 cannot learn a valid policy after 7 hours.

Figure 3 compares k-BFWS, with the same configuration as above, and PPO2 over the *Acrobot* benchmark. The reward function for Acrobot is such that every transition receives -1, but those reaching the goal, which attain a reward of 0. OpenAI implementation of the *Acrobot* environment has a built-in limit of 500 steps, so any policy achieving an expected reward of less than -500 is reaching the goal at least once. In this case, both PPO2 and k-BFWS reach the goal, yet PPO2 finds a more efficient policy. Indeed, k-BFWS with $f = \langle w \rangle$ is not optimal. We tried a different evaluation function, $f' = \langle w, g \rangle$, that takes into account accumulated rewards g, but then runtime per action became several times bigger than the duration of the control cycle Δt. The reason for this is that since the rewards attained are uniformly equal to -1, no guidance is provided by incorporating accumulated rewards to f'. On the other hand, PPO2 takes about 20 times longer to find a policy with the same quality as the plan found by k-BFWS.

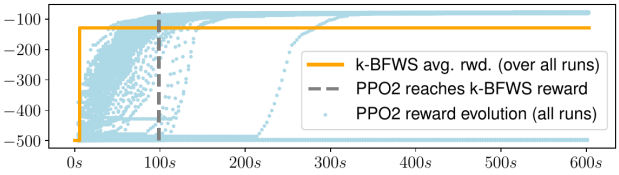

Figure 3: PPO2's reward evolution compared with k-BFWS-BEE on Acrobot (100 runs).

4.2 Stabilization Problems

We discuss now the *Cart Pole* and *Pendulum* benchmarks, where the task is to reach a given pose and maintain it stable for as long as possible, ideally, all the way to the end of the simulation. Figure 4 compares PPO2 and k-BFWS on *Cart Pole*. OpenAI implementation of the environment limits episodes to consist of 200 steps. From the Figure, we can see that k-BFWS is optimal, as it stabilizes the robot through the entire simulation. Also, compared with the runtime k-BFWS requires to find a plan, PPO2 takes 3 times more time to learn a policy with the same quality. This was an unexpected result, since the BEE features are designed to favor exploration, and in principle, risky behavior. This seemed to us to be a handicap for this task, since stabilizing the cart pole requires to avoid dead-ends, the states where the pole is in contact with the ground. Interestingly, those dead-ends were actually handled as negative goals by k-BFWS that, if discovered quickly, drive the planner away from selecting dangerous actions. We also ran RIW, limiting its computational budget on *Cart Pole* and selecting the action that leads to the most rewarding state once the budget is completed. As seen in Figure 5, RIW is able to stabilize the robot over the entire simulation if the decision time limit is set to 200ms. Since the control cycle for *Cart Pole* in OpenAI's implementation is 200ms, RIW attains very good performance and selects actions in "real time", that is, runtime per call is less than Δt, the duration of the control cycle of the simulator. Figure 6 shows how PPO2 compares with k-BFWS using BEE on the *Pendulum* benchmark. *Pen-*

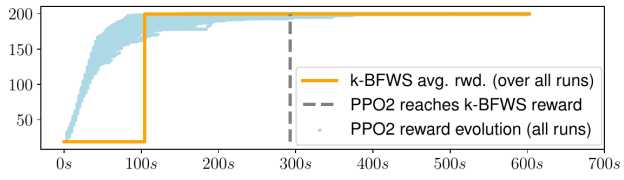

Figure 4: PPO2's reward evolution compared with k-BFWS-BEE on CartPole (100 runs).

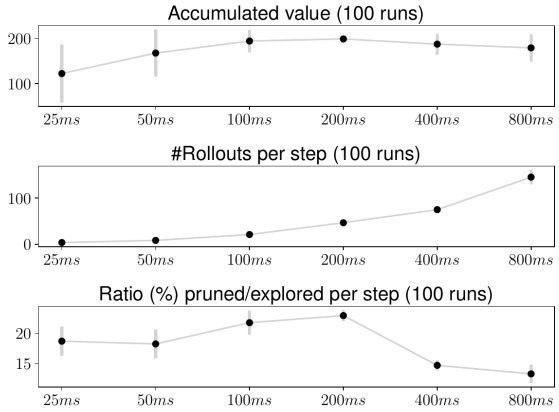

Figure 5: RIW performance for various decision time limits on *Cart-Pole*. Each point averages metric over 100 runs, vertical bars correspond to standard deviation.

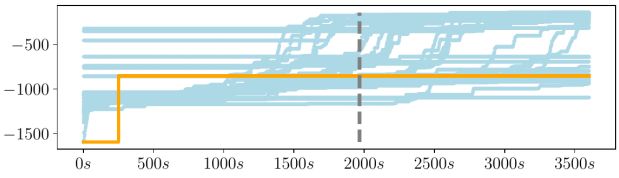

Figure 6: PPO2's reward evolution compared with k-BFWS-BEE on Pendulum (50 runs). The legend is the same as in Figure 3.

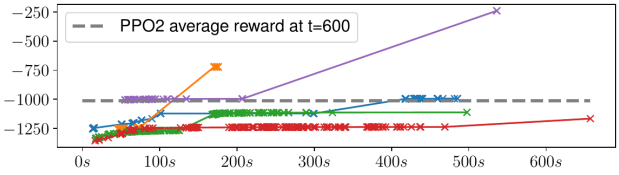

Figure 7: Anytime k-BFWS rewards with 10 minutes timeout on Pendulum. We show the 5 runs, out of 30, with the greatest improvement in the quality of plans.

dulum action space is continuous so we used a naive mapping into 12 different discrete actions. This branching factor is significantly higher than in other benchmarks, and presents a significant challenge to k-BFWS. In this experiment we used $f = \langle -d, g \rangle$, where d is the number of steps from s_0, breaking ties with the accumulated reward g. This effectively turns k-BFWS into a depth-first search algorithm. We observe that PPO2 takes 10 times longer than k-BFWS to find a policy of the same quality. However, a significant fraction of PPO2 runs greatly outperform k-BFWS as we let the clock run.

Figure 7 depicts the behaviour of *anytime* k-BFWS, using the same evaluation functions as above. For each run it is shown the time at which a better plan was found, and we compare it with the quality of the policy found by PPO2 after 10 minutes. We ran k-BFWS 30 times in this fashion, and in 15 occasions k-BFWS found plans superior to the baseline policy. These two observations suggest to us that warm-starting PPO2 with the best plan found by k-BFWS for a suitably set time-out, should be a useful technique to speed up PPO2 convergence to high quality policies.

4.3 Comparison with LQR Controllers

We have constructed two Linear Quadratic Regulator (LQR) controllers for Acrobot and Cartpole, following the method described by Perez et al. [2012]. We implemented it using off-the-shelf the components already available in the `Drake` framework [Tedrake and others, 2019]. The LQR controller obtains an average reward of -384.85 ± 198.4 in Acrobot, and of 200 ± 0 in Cartpole. While the later performance is the same as that of k-BFWS, the LQR controller is significantly worse on Acrobot, due to the perturbation of the initial states, that the controller cannot stabilise.

4.4 Recursive BEE Features

In our experiments we also considered an extension of the BEE features, to allow for them to recursively refine boundary extensions up to a given maximum level p. Each of the boundaries in Definition 3.1 becomes a node in a tree rooted at x_{s_0}, mapping values of state variables to a tuple of size p with the indices of the children of each boundary in the tree visited, as the value of a state variable is tested for inclusion.

We tested with RIW the effectiveness of the recursive BEE features with $p > 1$ on the Pendulum benchmark. For each value of p, ranging from 1 to 20, we ran RIW 10 times, measuring the average and standard deviation of the cost of each trajectory. We observed that while the average of costs improves for certain values of p greater than 1, the variance in the results is very high, and it cannot be ruled out that the actual means are significantly different. Similar results were obtained on *Cart Pole*, *Mountain Car* and *Acrobot*. This was a surprising result, and we look forward to develop more benchmarks in order to rule out the possibility that these observations are a characteristic specific to these benchmarks.

5 Related Work

A similar issue to the one exposed in Section 1 affects classical reinforcement learning algorithms too, such as the tabular form of Q-Learning [Sutton and Barto, 2018]. In response, a number of so–called *tile coding* techniques were proposed and tested on the very same environments we discuss in Section 4 [Sutton, 1996], with most tile coding schemes proposed assuming the number and geometry of tiles to be fixed. We discuss two proposed schemes where tiles were defined dynamically during learning. Whiteson et al. [2007] introduced a scheme where tiles were split into smaller regular ones, triggered by the value function learning loss plateauing. Heuristics were proposed that tried to optimize splits so as to maximize the magnitude of the updates of the value function. Lin and Wright [2010] proposed a more flexible scheme, called *Evolutionary Tile Coding* (EvoTC), where tiles are not required to have regular sizes and are organised hierarchically. The key difference between our approach and Lin's is that the later uses a genetic algorithm, off–line, to optimize the tile encoding using as fitness the performance of the learning algorithm.

6 Conclusions & Future Work

We have demonstrated the proposed feature encoding, in conjunction with the two main width-based search algorithms, to have performance superior or comparable to state-of-the-art DRL algorithms and applications of optimal control theory. We look forward to extend this research by looking at two possible applications of our planner. One is to provide an *excitation strategy* for system identification & model learning [Mitrovic *et al.*, 2010] more effective than *motor babbling* or random walks. The other one is to use our planners to initialize non–linear optimal control solvers with a *nominal trajectory*, speeding up convergence to high quality solutions [Nocedal and Wright, 2006].

Acknowledgements

F. Teichteil-Königsbuch was supported by Airbus R&T.
M. Ramirez and N. Lipovetzky wish to acknowledge the support of the DST Group via research contract 9156.

References

[AIRBUS - AI Research, 2020] AIRBUS - AI Research. scikit-decide library. https://github.com/airbus/scikit-decide, 2020.

[Bandres *et al.*, 2018] Wilmer Bandres, Blai Bonet, and Hector Geffner. Planning with pixels in (almost) real time. In *AAAI*, 2018.

[Bertsekas, 2017] Dmitri P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 4th edition, 2017.

[Bonet and Geffner, 2013] Blai Bonet and Hector Geffner. *A Concise Introduction to the Models and Methods for Automated Planning*. Morgan & Claypool, 2013.

[Boyan and Moore, 1995] Justin A. Boyan and Andrew W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In *NIPS*, 1995.

[Frances *et al.*, 2017] Guillem Frances, Miquel Ramírez Jávega, Nir Lipovetzky, and Hector Geffner. Purely declarative action descriptions are overrated: Classical planning with simulators. In *IJCAI*, 2017.

[Geffner and Geffner, 2015] Tomás Geffner and Hector Geffner. Width-based planning for general video-game playing. In *AIIDE*, pages 23–29, 2015.

[Gerevini *et al.*, 2019] Alfonso E. Gerevini, Nir Lipovetzky, Francesco Percassi, Alessandro Saetti, and Ivan Serina. Best-first search for multi agent privacy-preserving planning. In *ICAPS*, 2019.

[Helmert *et al.*, 2008] Malte Helmert, Minh Do, and Ioannis Refanidis. Ipc 2008 deterministic competition. In *6th Int'l Planning Competition Booklet (ICAPS-08)*, 2008.

[Helmert, 2002] Malte Helmert. Decidability and undecidability results for planning with numerical state variables. In *AIPS*, 2002.

[Jinnai and Fukunaga, 2017] Yuu Jinnai and Alex Fukunaga. Learning to prune dominated action sequences in online black-box planning. In *AAAI*, pages 839–845, 2017.

[Keyder and Geffner, 2009] Emil Keyder and Hector Geffner. Soft goals can be compiled away. *Journal of Artificial Intelligence Research*, 36:189–211, 2009.

[Lin and Wright, 2010] Stephen Lin and Robert Wright. Evolutionary tile coding: An automated state abstraction algorithm for reinforcement learning. In *AAAI*, 2010.

[Lipovetzky and Geffner, 2012] Nir Lipovetzky and Hector Geffner. Width and serialization of classical planning problems. In *ECAI*, pages 540–545, 2012.

[Lipovetzky and Geffner, 2017a] Nir Lipovetzky and Hector Geffner. Best-first width search: Exploration and exploitation in classical planning. In *AAAI*, 2017.

[Lipovetzky and Geffner, 2017b] Nir Lipovetzky and Hector Geffner. A polynomial planning algorithm that beats lama and ff. In *ICAPS*, 2017.

[Lipovetzky *et al.*, 2015] Nir Lipovetzky, Miquel Ramirez, and Hector Geffner. Classical planning with simulators: results on the atari video games. In *IJCAI*, 2015.

[Mitrovic *et al.*, 2010] Djordje Mitrovic, Stefan Klanke, and Sethu Vijayakumar. Adaptive optimal feedback control with learned internal dynamics models. In Olivier Sigaud and Jan Peters, editors, *From Motor Learning to Interaction Learning in Robots*. Springer, 2010.

[Moore, 1990] A. Moore. *Efficient Memory–Based Learning for Robot Control*. PhD thesis, University of Cambridge, 1990.

[Nocedal and Wright, 2006] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, 2nd edition, 2006.

[O'Toole *et al.*, 2019] Stefan O'Toole, Miquel Ramirez, Nir Lipovetzky, and Adrian Pearce. Width-based lookaheads augmented with base policies for stochastic shortest paths. In *ICAPS Workshop on Heuristics and Domain Independent Planning*, 2019.

[Perez *et al.*, 2012] Alejandro Perez, Robert Platt Jr., George Konidaris, Leslie Kaebling, and Tomas Lozano-Perez. Lqr-rrt*: Optimal sampling-based motion planning with automatically derived extension heuristics. In *ICRA*, 2012.

[Ramirez *et al.*, 2018] Miquel Ramirez, Michael Papasimeon, Nir Lipovetzky, Lyndon Benke, Tim Miller, Adrian R Pearce, Enrico Scala, and Mohammad Zamani. Integrated hybrid planning and programmed control for real time uav maneuvering. In *AAMAS*, 2018.

[Schulman *et al.*, 2017] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[Shleyfman *et al.*, 2016] Alexander Shleyfman, Alexander Tuisov, and Carmel Domshlak. Blind search for atari-like online planning revisited. In *IJCAI*, pages 3251–3257, 2016.

[Stepanov and McJones, 2009] Alexander Stepanov and Paul McJones. *Elements of Programming*. Addison-Wesley, 2009.

[Stevens *et al.*, 2015] Brian L. Stevens, Frank L. Lewis, and Eric N. Johnson. *Aircraft Control and Simulation: Dynamics, Controls Design and Autonomous Systems*. John Wiley & Sons, 2015.

[Sutton and Barto, 2018] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2nd edition, 2018.

[Sutton, 1996] Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *NIPS*, 1996.

[Tedrake and others, 2019] Russ Tedrake et al. Drake: Model-based design and verification for robotics, 2019.

[Whiteson *et al.*, 2007] Shimon Whiteson, Matthew E. Taylor, and Peter Stone. Adaptive tile coding for value function approximation. Technical report, Computer Science Department, University of Texas at Austin, 2007.

[Willems, 2007] Jan C. Willems. The behavioral approach to open and interconnected systems. *IEEE Control Systems Magazine*, 27(6):46–99, 2007.