**IEEE Control Systems Letters paper presented at**
**2023 American Control Conference (ACC)**
**San Diego, CA, USA. May 31 - June 2, 2023**

# GrAVITree: Graph-based Approximate Value Function In a Tree

Patrick H. Washington, David Fridovich-Keil, and Mac Schwager

***Abstract*— In this paper, we introduce GrAVITree, a tree- and sampling-based algorithm to compute a near-optimal value function and corresponding feedback policy for indefinite time-horizon, terminal state-constrained nonlinear optimal control problems. Our algorithm is suitable for arbitrary nonlinear control systems with both state and input constraints. The algorithm works by sampling feasible control inputs and branching backwards in time from the terminal state to build the tree, thereby associating each vertex in the tree with a feasible control sequence to reach the terminal state. Additionally, we embed this stochastic tree within a larger graph structure, rewiring of which enables rapid adaptation to changes in problem structure due to, e.g., newly detected obstacles. Because our method reasons about global problem structure without relying on (potentially imprecise) derivative information, it is particularly well suited to controlling a system based on an imperfect deep neural network model of its dynamics. We demonstrate this capability in the context of an inverted pendulum, where we use a learned model of the pendulum with actuator limits and achieve robust stabilization in settings where competing tree-based and derivative-based techniques fail.**

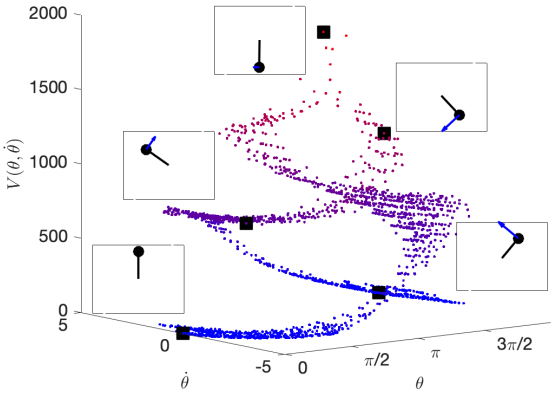


**Fig. 1:** An example of a tree corresponding to pendulum swingup. The tree is constructed backward-in-time. Insets show position and velocity at illustrative states.

## I. Introduction

It is generally difficult to find global solutions to nonlinear optimal control problems, and it can be particularly difficult to do so in the presence of actuation and state constraints. Most existing approaches rely on either approximate solutions to the Hamilton-Jacobi-Bellman (HJB) equation or settle for locally-convergent model predictive control (MPC) techniques. HJB and path integral methods provide an implicit state feedback policy; however, they are computationally prohibitive in the general case. Indeed, HJB methods are often only effective when they can be computed offline, and are hence not well suited to settings in which problem structure changes at run-time, e.g. due to a new obstacle. Conversely, nonlinear MPC methods can react online to such changes, but do not offer a feedback policy, and can suffer from impractically long optimization times for nonlinear dynamics, non-convex constraints and high state dimensions. Furthermore, nonlinear MPC cannot give optimality guarantees for solution trajectories in such non-convex cases. Most importantly, solvers for nonlinear MPC use Jacobians, and sometimes Hessians, of the dynamics model [1], which can be quite unreliable in deep learned dynamics models, leading to poor control performance.

We propose a sampling-based solution technique, called **Gr**aph-based **A**pproximate **V**alue Function **I**n a **Tree** (GrAVITree), to approximately solve the HJB equation for a class of constrained nonlinear optimal control problems. Our method avoids predefined, often axis-aligned finite element grids common to other HJB solution methods, and, unlike other HJB methods, maintains a graph of multiple possible paths to a goal state to enable fast online "rewiring" in response to changing constraint structure. Furthermore, our method does not rely on gradients of the dynamics model, making it suitable for controlling systems whose dynamics are modeled by a deep neural network.

As illustrated in Fig. 1, we construct a tree incrementally *backward in time* from a desired terminal state and rewire the tree such that trajectories through the tree are optimal given the discrete set of state samples in the graph. As the samples fill the dynamically-feasible submanifold of the state space, we observe that optimal trajectories through the tree approach globally optimal trajectories in the continuous state space. Critically, at runtime, we also provide a feedback control law that uses this tree and the current system state to interpolate an input at *any* state, even one which was not sampled in the tree.

P. Washington and M. Schwager are with the Department of Aeronautics & Astronautics, Stanford University. D. Fridovich-Keil is with the Department of Aerospace Engineering, UT Austin. Correspondence to phw@stanford.edu.

Toyota Research Institute provided funds to support this work. The NASA University Leadership initiative (grant #80NSSC20M0163) provided funds to assist the authors with their research, but this article solely reflects the opinions and conclusions of its authors and not any NASA entity. The first author was supported on a National Defense Science and Engineering Graduate (NDSEG) Fellowship. We are grateful for this support.

## II. Problem Formulation

### A. Optimal control problem

Consider the following optimal control problem which is terminal state constrained and has unspecified final time:

$$J(x_0;\hat{x}) = \min_{\mathbf{u},t'\geq 0} \sum_{t=0}^{t'} g(x_t,u_t) \tag{1}$$

$$\text{subject to } u_t \in \mathcal{U} \subseteq \mathbb{R}^m \tag{2}$$

$$f^+(x_t,u_t) = x_{t+1} \in \mathcal{X} \subseteq \mathbb{R}^n \tag{3}$$

$$x_{t'} = \hat{x}. \tag{4}$$

Here, the system dynamics are given in (3) for state $x$ and input $u$. The desired terminal state is $\hat{x}$, which must occur at some future time step $t'$ (without loss of generality, we presume that time starts at $t = 0$). Further, we presume that state and input are uniformly bounded within compact sets $\mathcal{X}$ and $\mathcal{U}$, respectively.

In (1), the decision variables are $\mathbf{u} := (u_0, \ldots, u_{t'})$ and the final time $t' \geq 0$ at which the terminal constraint (4) must be satisfied. The objective is then the accumulated running cost $g$. Certain problems of this form are well-studied. For example, when $g(x,u) \equiv 0$, (1) reduces to a standard reachability problem, for which level-set methods [2, 3] are available for low-dimensional systems (typically, $n \leq 5$). Other methods exist as well, such as ellipsoidal [4, 5] and zonotope [6] approximations. However, as we are concerned with the general case in which $g \not\equiv 0$ and for larger state space dimensions $n$, our approach does not explicitly attempt to either (a) solve a fixed-horizon HJB equation, or (b) build a geometric approximation to any reachable set of interest.

These (and other, still more general) types of problems are commonly solved to local optimality using a variety of nonlinear programming techniques, e.g., [7, 8]. Unlike these methods, we focus on approximating a global optimum and an associated state-feedback strategy (which is typically beyond the scope of nonlinear programming and MPC-based approaches).

### B. Backward dynamics

So far, we have presumed that system dynamics are represented in discrete time by (3). Later, we shall make use of the *backward* form of the dynamics. Before defining the backward dynamics, we first note that continuous-time dynamics, defined by $\dot{x}_\tau = F(x_\tau, u_\tau)$, are easily discretized. The discrete-time (forward) dynamics $f^+$ from (3) can be written as

$$x_{t+1} = f^+(x_t,u_t) = x_t + \int_{\tau(t)}^{\tau(t+1)} F\big(x(\tau),u(\tau)\big)d\tau, \tag{5}$$

where $\tau$ represents the continuous-time value, $\tau(t)$ is the time associated with time step $t$, and state/input signals are indicated by $x(\cdot)$ and $u(\cdot)$, respectively. The form of (5) suggests the following equation for backward dynamics $f^-$:

$$x_{t-1} = f^-(x_t,u_{t-1}) = x_t + \int_{\tau(t)}^{\tau(t-1)} F\big(x(\tau),u(\tau)\big)d\tau. \tag{6}$$

Practically, we will approximate the integrals in (5) and (6) via numerical quadrature, e.g.,

$$\int_{\tau_1}^{\tau_2} F(x_\tau,u_\tau) \approx \sum_{k=0}^{\lceil(\tau_2-\tau_1)/\delta\tau\rceil} F(x_{\tau_1+k\delta\tau},u_{\tau_1+k\delta\tau})\delta\tau \tag{7}$$

where $\delta\tau \ll 1$ is a small time interval.

Later, in Sec. IV, we will use the backward form of dynamics (6) extensively to construct a tree whose branches encode trajectories which terminate in the desired state $\hat{x}$.

## III. Related Work

In principle, our work is closely related to classical dynamic programming approaches from the optimal control literature. Algorithmically, however, it draws significant inspiration from related work in sample-based kinodynamic motion planning. Both areas are extremely well-studied; what follows is a succinct summary of existing techniques and a discussion of how they relate to the present work.

### A. Dynamic programming

Problem (1) may be viewed as a *shortest-path problem*, and hence admits a dynamic programming solution to compute the optimal path length, or *value function*, $V(\cdot)$ for every state $x$. This fact forms the basis for a wide variety of techniques; the interested reader is directed to [9]. Exact dynamic programming methods scale exponentially with the size of discrete state spaces and require discretization to extend to continuous problems. Approximate methods [10] are increasingly popular and some come with formal convergence guarantees [11], however most methods do not come with such guarantees and computational complexity can still be a challenge with these methods. While they are still quite successful in a number of applications [12], two areas of particular note are linear-quadratic problems (and approximations) such as [13–15] and Monte Carlo Tree Search (MCTS) methods, which have been used extensively to solve Partially Observed Markov Decision Processes (POMDPs) [16, 17].

Reinforcement learning [18] approaches form a broad class of approximate dynamic programming methods that has received significant attention in recent years. "Deep" variants of these methods refine a deep learned representation of the value function (or implicitly represent it with a *policy* or feedback law). Experimental results in this field are extremely encouraging, e.g. in the domains of arcade games [19], Go [20], robotic arm control [21], and dexterous manipulation [22]. However, despite these experimental successes and several initial attempts [23–25], it is still uncommon for reinforcement learning methods to consider non-dynamic constraints of the form (2) and (4). Our method accounts for these types of constraints by construction.

### B. Sample-based kinodynamic planning

Sample-based motion planning has also been extraordinarily successful in recent decades. Early approaches such as the well-known rapidly-exploring random tree (RRT) [26] and probabilistic roadmap (PRM) [27] have since been generalized to work in kinodynamic settings [28, 29] and guarantee asymptotic optimality [30, 31]. In these algorithms, states are sampled at random from the feasible set $\mathcal{X}$ and added to a graph structure which encodes the proximity of sampled states. Most methods construct *tree* structures from the current state, the desired terminal state, or both [32]. Sampling methods can (a) scale to CPU-constrained [33] and high-dimensional systems [34, 35] and (b) execute in real-time for modest problem instances [36]. Additionally, sampling-based planning can be coupled with nonlinear stability analysis to provide runtime stability guarantees [37]. Although the present work also defines a feedback control law, it does not rely upon solving sum-of-squares programs (and is hence more scalable, but does not guarantee stability).

Our own work is inspired by the RRT* algorithm of [30]; in particular, the `UpdateTree` method (Sec. IV-B) is a direct extension of the `rewire` procedure from RRT* to our setting. Further, RRT* presumes the existence of a `steer` subroutine to drive a dynamical system to a desired state. In general, this is a difficult problem, and in the same spirit as [38] we contribute an efficient, general method for approximate steering in Sec. IV-A. Unlike RRT*, however, we build our tree *backward* from the terminal state since we presume that only it is fixed and the initial state will be determined at runtime. Additionally, we also provide a feedback law that drives arbitrary states, not just those in the tree, to the goal state. Hence, our method is robust to some degree of model mismatch, where the dynamics of the system $F$ differ from those of the physical system.

## IV. GRAVITREE

We solve problem (1) by constructing a graph $\mathcal{G} = (\mathcal{V}_G, \mathcal{E}_G)$ with vertices $\mathcal{V}_G$ and edges $\mathcal{E}_G$, where vertices contain state information and edges contain the control and stage cost information. The graph is directed and may have cycles.

For some vertex $v_G \in \mathcal{V}_G$, denote the state with $v_G[x]$. For each edge $e_G \in \mathcal{E}_G$, denote the control with $e_G[u]$ and the stage cost with $e_G[c]$.

More precisely, if there is an edge $e_G^{ij}$ that connects vertex $v_G^i$ to vertex $v_G^j$, then

$$v_G^j[x] = f^+(v_G^i[x], e_G^{ij}[u]). \tag{8}$$

We will construct the tree backward in time; hence,

$$v_G^i[x] = f^-(v_G^j[x], e_G^{ij}[u]). \tag{9}$$

Moreover, the stage cost of an edge is

$$e_G^{ij}[c] = g(v_G^i[x], e_G^{ij}[u]). \tag{10}$$

While the graph $\mathcal{G}$ is constructed, we also build a tree $\mathcal{T} = (\mathcal{V}_T, \mathcal{E}_T)$. Unlike $\mathcal{G}$, the tree's vertices contain both state and cost-to-go information, and its edges contain controls and the cost of executing those controls from the corresponding states. Importantly, where $\mathcal{G}$ contains edges from each vertex to all vertices determined to be reachable in one step, in $\mathcal{T}$ each vertex has a single outgoing edge corresponding to the best control action to take from that state, according to the current cost-to-go information.

For each vertex $v_T \in \mathcal{V}_T$, denote the state with $v_T[x]$ and the cost-to-go with $v_T[J]$. For each edge $e_T \in \mathcal{E}_T$, denote the control with $e_T[u]$ and the stage cost with $e_T[c]$. As in graph $\mathcal{G}$, tree vertices satisfy

$$v_T^j[x] = f^+(v_T^i[x], e_T^{ij}[u]),\ v_T^i[x] = f^-(v_T^j[x], e_T^{ij}[u]) \tag{11}$$

and the stage costs encoded in the tree are identical to the those encoded in the graph:

$$e_T^{ij}[c] = g(v_T^i[x], e_T^{ij}[u]). \tag{12}$$

From the time-additive form of (1), the cost-to-go of a vertex is determined by adding the stage cost to the cost-to-go of the parent vertex:

$$v_T^i[J] = e_T^{ij}[c] + v_T^j[J]. \tag{13}$$

The root of the tree has $v_T^{\text{root}}[x] \equiv \hat{x}$ and $v_T^{\text{root}}[J] \equiv 0$.

Tree vertices are in one-to-one correspondence with graph vertices. Meanwhile, every edge in $\mathcal{T}$ has a corresponding edge in $\mathcal{G}$, but not every edge in $\mathcal{G}$ has a corresponding edge in $\mathcal{T}$. Consequently, it is convenient to represent both $\mathcal{G}$ and $\mathcal{T}$ with a single data structure; however, they serve different purposes and are best thought of separately. Additionally, note that despite building backwards, the direction of the edges in both the tree and the graph is always defined such that applying control $e^{ij}[u]$ from state $v^i[x]$ yields state $v^j[x]$.

Construction proceeds according to Alg. 1. First, a random state is sampled from the state space and the closest vertex by L2-norm in the graph is identified, similar to RRT [26]. This method of choosing a vertex encourages even growth through the state space. Sampling from the vertices themselves empirically leads to uneven coverage of the space. Second, several controls are randomly sampled and applied backward in time from the chosen vertex. The control that leads to the state that is furthest from any existing vertex in the graph is chosen. We do this to prioritize even coverage of the state space but other control selection techniques, such as targeting the sampled state to prioritize certain regions of the space, are possible as well. Third, a vertex and an edge that correspond to that state and control are added to the graph and the tree. Fourth, edges from the new vertex to other existing vertices are identified. This procedure is detailed in Sec. IV-A. Finally, the tree is updated using the added vertex and new edges in the graph, detailed in Sec. IV-B.

### A. Find Connections

This portion of the algorithm aims to identify potential parent and child vertices for a given vertex in the graph. Allowing a vertex to select a new parent provides an opportunity for it to improve its cost-to-go. Finding new potential

**Algorithm 1:** Build($\hat{x}$,$f^+$,$f^-$,$g$)

Initialize graph $\mathcal{G}$ with vertex $(\hat{x})$
Initialize tree $\mathcal{T}$ with vertex $(\hat{x}, 0)$
**while** *not finished* **do**
  Sample state $x_s \in \mathcal{X}$
  Pick vertices $v_G, v_T$ closest to $x_s$
  Choose control $u$ to apply from $v_G, v_T$
  $x \leftarrow f^-(v_G[x], u)$
  $c \leftarrow g(x, u)$ ▷ stage cost
  $v'_G \leftarrow (x)$ ▷ graph vertex
  $e_G^{v'_G v_G} \leftarrow (u, c)$ ▷ graph edge
  Add $v'_G, e'_G$ to $\mathcal{G}$
  $v'_T \leftarrow (x, c + v_T[J])$ ▷ tree vertex
  $e'_T \leftarrow (u, c)$ ▷ tree edge
  Add $v'_T, e'_T$ to $\mathcal{T}$
  FindConnections($v'_G, \mathcal{G}, \mathcal{T}$)
  UpdateTree($v'_G, v'_T, \mathcal{G}, \mathcal{T}$)
**return** $\mathcal{G}, \mathcal{T}$

children allows other vertices to choose it as a parent and similarly reduce their own cost-to-go.

First, controls are sampled one step forward in time from the new vertex to identify other potential parents. This works by simulating the controls from that vertex, identifying the vertices closest to the resulting states, and attempting to refine the controls so that the two coincide. To avoid issues related to the imprecision of gradients of (potentially neural network) dynamics, we employ a derivative-free optimization technique based upon the pattern search of [39], in which we use several random directions instead of only the coordinate axes. For each search that reaches some threshold of the targeted vertex, an edge is added to the graph. Second, this process is repeated for controls going backward in time to identify potential children.

**Algorithm 2:** FindConnections($v_G, \mathcal{G}, \mathcal{T}$)

**for** $v'_G \in$ *Candidate Parents* $\subset \mathcal{V}_G$, $v'_G \neq v_G$ **do**
  **if** $\exists u$ *s.t.* $\|f^+(v_G[x], u) - v'_G[x]\| < \varepsilon$ **then**
    $e_G^{v_G v'_G} \leftarrow (u, g(v_G[x], u))$
**for** $v'_G \in$ *Candidate Children* $\subset \mathcal{V}_G$, $v'_G \neq v_G$ **do**
  **if** $\exists u$ *s.t.* $\|f^-(v_G[x], u) - v'_G[x]\| < \varepsilon$ **then**
    $e_G^{v'_G v_G} \leftarrow (u, g(v'_G[x], u))$

### B. Update Tree

Once a vertex has determined its potential parents and children from the set of current vertices in the graph, it must update its tree information. First, it looks at its potential parents and decides whether any result in lower cost-to-go. Second, the vertex checks on its potential children and informs them of its cost-to-go. If a potential child can improve its cost-to-go by descending from the current vertex, then the `UpdateTree` routine is run on that child. It is important to note that since vertices may have many potential parents, the `UpdateTree` routine may be run on the same vertex multiple times. However, it will only continue with the recursion along that path through the graph if there is improvement. This prevents cycles through the graph since a path that goes through the same vertex twice cannot be an improvement, preventing infinite recursion.

**Algorithm 3:** UpdateTree($v_G, v_T, \mathcal{G}, \mathcal{T}$)

**for** $e_G^{v_G i} \in \mathcal{E}_G$ **do**
  $v'_G \leftarrow$ graph vertex corresponding to $i$
  $v'_T \leftarrow$ tree vertex corresponding to $v'_G$
  **if** $e_G^{v_G v'_G}[c] + v'_T[J] < v_T[J]$ **then**
    Remove current edge starting at $v_T$ from $\mathcal{T}$
    $e_T^{v_T v'_T} \leftarrow (e_G^{v_G v'_G}[u], e_G^{v_G v'_G}[c])$
    Add $e_T^{v_T v'_T}$ to $\mathcal{T}$
    $v_T[J] \leftarrow e_T^{v_T v'_T} + v'_T[J]$
**for** $e_G^{i v_G} \in \mathcal{E}_G$ **do**
  $v'_G \leftarrow$ graph vertex corresponding to $i$
  $v'_T \leftarrow$ tree vertex corresponding to $v'_G$
  **if** $e_G^{v'_T v_T}[c] + v_T[j] < v'_T[J]$ **then**
    UpdateTree($v'_G, v'_T, \mathcal{G}, \mathcal{T}$)

### C. Termination Check

As with most graph-based planning methods [26, 30], our method is *anytime* and there exist a number of reasonable stopping criteria. Options include spending a fixed maximum wall-clock time, waiting until the tree contains at least a certain number or density of vertices, and halting when the tree includes a vertex near a particular location of interest. The corresponding planning algorithm does not fundamentally change if the stopping condition changes. In Sec. VII, we use a time limit.

### D. Cost-to-Go Properties

In this section we state and prove two fundamental properties related to the cost-to-go represented in GrAVITree. In future work, we intend to extend these results to establish convergence guarantees for the value function across the continuous state space. Let $J^*(x)$ be the optimal cost-to-go from state $x$ and $v_T[J]^k$ be the cost-to-go at iteration $k$.

*Proposition 1 (Conservative Cost-to-Go):* The cost-to-go at each vertex in the tree is a conservative estimate for the optimal cost-to-go, i.e., $v_T[J]^k \geq J^*(v_T[x]),\ \forall v_T \in \mathcal{T},\ \forall k \geq 0$.

*Proof:* The cost-to-go estimate at a vertex $v_T$ is computed from a path of dynamically feasible states through the graph to the goal. Therefore, the optimal cost-to-go, considering all feasible paths in the continuous space to the goal, must be less than or equal to this estimate. ■

*Proposition 2 (Decreasing Cost-to-Go):* The cost-to-go at every vertex in the tree is monotonically

non-increasing as new vertices are added, i.e., $v_T[J]^{k+1} \leq v_T[J]^k,\ \forall v_T \in \mathcal{T}, \forall k \geq 0$.

*Proof:* This is because (1) vertex states and graph edge controls do not change and (2) the edges in the tree are only changed when a new edge improves the cost-to-go. ■

*Remark 1:* As is typical in sampling based planning, Prop. 1 rests on the assumption that vertices in the graph are exactly connected by dynamically feasible edges. In implementation, connecting these edges requires a numerical optimization subject to a finite tolerance, specifically $\epsilon$ in Alg. 2. With $\epsilon$ sufficiently small, we observe that Prop. 1 holds in practice despite these numerical errors.

## V. Modifying the Problem at Runtime

A key benefit of storing both the graph and the tree is that the tree can be recomputed online with very little effort in response to modifications to the problem parameters. This can include changing the goal state, changing the stage cost function, and/or adding new state and/or control constraints.

To do so, we first update the cost information in the edges of the graph to reflect the new problem structure. Second, we discard the tree and replace it by finding the shortest path, as measured by number of edges traversed, through the graph to the goal from each vertex, ignoring costs. Third, we repeatedly iterate through each vertex, identifying better parents, as measured by cost-to-go, until no vertex changes in a given cycle. We must iterate in this way and not according to Alg. 3 because here, there is no guarantee that the upstream vertices are optimal until the iterations stop.

Note that reachability will not change if only the cost function changes. However, changing the goal or adding new constraints can make it so that some vertices no longer have a valid path through the graph to the goal. For example, if there is now a wall partitioning the state space, vertices on one side of the wall cannot reach the other side. Additionally, if an edge starts at a state or uses a control that violates a constraint, the stage cost of that edge is set to infinity. By preserving (and not pruning) vertices and edges which violate the new constraints, we retain information which may be useful if the problem is modified further.

Because tree reconstruction is relatively efficient, we construct the graph only once for the least-constrained version of the problem. After that, constraints may be added, e.g., as information regarding obstacles is gathered from onboard sensors.

## VI. Control

The tree $\mathcal{T}$ serves as the basis for controlling the system since it encodes the value function for the system and control information. Each vertex contains a point in the state space and the corresponding cost-to-go, while the edges contain control actions that lead to the root of the tree (i.e., the goal). Of course, while controlling the system, the state will never perfectly match the state in any of the vertices, so some interpolation technique is required.

**Algorithm 4:** ModifyProblem

```
for e_G^{ij} ∈ E_G do
    e_G[c] ← g(v_G^i[x], e_G^{ij}[u])
for v_G ∈ V_G do
    v_T ← (v_G[x], ∞)
    e_T ← the e_G that gives the shortest path
changed ← true
while changed do
    changed ← false
    for v_T^i, e_T^{ij} ∈ T do
        if e_T^{ij}[c] + v_T^j[J] < v_T^i[J] then
            v_T^i[J] ← e_T^{ij}[c] + v_T^j[J]
            changed ← true
```

In Sec. VII, we use a bump function, defined by

$$\varphi(x, x') = \begin{cases} \exp\left(-\frac{1}{1-(\gamma\|x-x'\|)^2}\right), & \|x - x'\| < \frac{1}{\gamma} \\ 0, & \text{otherwise,} \end{cases} \tag{14}$$

to interpolate the value data in the tree. To generate a control at each time step, we perform a pattern search [39] to find the control that minimizes the sum of the stage cost and the value of the subsequent state:

$$u^* = \arg\min_{u \in \mathcal{U}} g(x, u) + \frac{\sum_{v_T \in \mathcal{T}} \varphi(f^+(x, u), v_T[x]) v_T[J]}{\sum_{v_T \in \mathcal{T}} \varphi(f^+(x, u), v_T[x])}. \tag{15}$$

**Algorithm 5:** Control($x$)

```
u ← 0                                              ▷ Vector in R^m
Initialize α ∈ R > 0
Initialize δ_1, ..., δ_{2m} with unit direction vectors in R^m
x'_0 ← f^+(x, u)
Estimate J(x'_0) using interpolation
J_best ← J(x'_0)
while α > ε do
    x'_k ← f^+(x, u + αδ_k) for k ∈ {1, ..., 2m}
    Estimate J(x'_k) using interpolation for each k
    k_min ← arg min_k J(x'_k)
    if J(x'_{k_min}) < J_best then
        J_best ← J(x'_{k_min}), u ← u + δ_{k_min}
    else
        α ← α/2
return u
```

## VII. Results

We demonstrate the capability of our approach on two systems: a two-dimensional single integrator and an inverted pendulum. These systems are informative demonstrations because state obstacles are natural to add to the former and control constraints lead to stark changes in the value function of the latter.

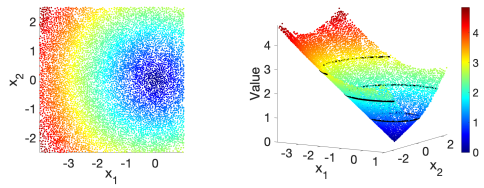


**Fig. 2:** Values in the tree for the unconstrained single integrator system. Black rings that represent level sets for distance to the goal are added to the right plot. The tree's values match the true shortest distance to the origin. This is the result of 1 minute of computation.

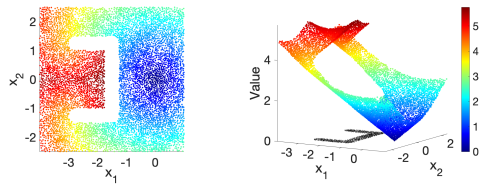


**Fig. 3:** Values in the tree for the constrained single integrator system. The empty region on the left plot shows the added obstacle. The points within the obstacle are shown as a shadow on the right. This data was adapted from the data in Fig. 2 in 1-2 seconds.

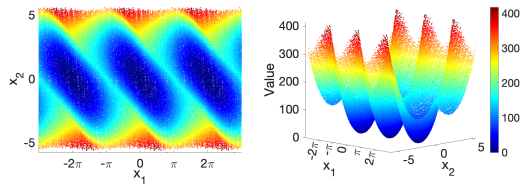


**Fig. 4:** Values in the tree for the inverted pendulum system. The value data is repeated in the $x_1$ direction to demonstrate continuity. This is the result of 30 minutes of computation.

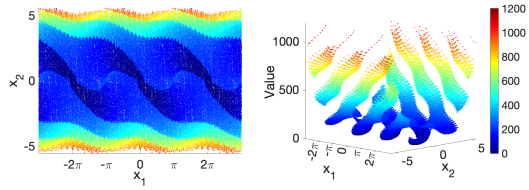


**Fig. 5:** Values in the tree for the inverted pendulum system with more restrictive control limits. The same graph was used. Only the optimal paths through the graph changed. Again, the value data is repeated along the $x_1$ direction to demonstrate continuity. This data was adapted from the data in Fig. 4 in less than 5 seconds.

### *A. Single Integrator and State Constraints*

The single integrator has two states and two control inputs which evolve linearly as

$$\dot{x} = F(x, u) = u, \tag{16}$$

and we presume a stage cost of

$$g(x, u) = \|u\|_2 \Delta t. \tag{17}$$

The terminal state is assumed to be $\hat{x} = 0$; hence, the objective is to take the shortest path to the origin.

The initial graph construction is done in the absence of obstacles. The value function for this is shown in Fig. 2. Without any additional graph building, we add a U-shaped obstacle that forces many paths to be longer. The value function after including the obstacle is shown in Fig. 3.

### *B. Inverted Pendulum and Control Constraints*

The inverted pendulum has two states and one control input, which follow the nonlinear model

$$\dot{x} = F(x, u) = \begin{bmatrix} x_2 \\ \sin(x_1) + u \end{bmatrix}, \tag{18}$$

where $x_1$ is wrapped to $[-\pi, \pi]$. Stage cost is given by

$$g(x, u) = x^\top x + u^\top u, \tag{19}$$

which encodes an objective of stabilizing at the upright (unstable) equilibrium. Given large enough control actuation limits, the inverted pendulum is feedback linearizable and hence can be treated similarly to the single-integrator. The value function for this setting is shown in Fig. 4. Indeed, it appears nearly quadratic as one would expect in an optimal control problem with linear dynamics and quadratic objective. However, when the control input is limited, the behavior is clearly nonlinear. As one expects, more restrictive control limits mean that the pendulum may need to swing back and forth to build momentum or take multiple rotations to bleed momentum. This is demonstrated in Fig. 5. There are distinct layers and spirals in the value function, where each layer/spiral indicates another oscillation that is required.

### *C. Controlling Learned Dynamics Models*

GrAVITree does not rely on having any knowledge of the structure of the dynamics equations. It only needs to be able to call blackbox functions corresponding to $f^+$ and $f^-$. This enables GrAVITree to control systems whose dynamics are described with a neural network, such as one learned from collected measurements of the true system.

Here, consider a system with

$$\dot{x} = F(x, u) = \text{Network}([x; u]) \approx \begin{bmatrix} x_2 \\ \sin(x_1) + u \end{bmatrix}, \tag{20}$$

where the network was trained on 50000 random ($x$, $u$, $\dot{x}$) triples from the inverted pendulum dynamics. However, there is inevitable mismatch between the learned model and true system, which means that following an open-loop plan is not sufficient. The mean error of the network over 10000 random samples is $|\Delta \dot{x}_1| \approx 0.064$ rad/s and $|\Delta \dot{x}_2| \approx 0.177$ rad/s$^2$.

We performed three simulated experiments for five different algorithms. First, we plan and execute using the physics model from (18). Second, we plan and execute using the network model from (20). Third, we plan using the network model and execute on the physics model. The five algorithms include GrAVITree implemented in C++ using the Eigen library [40], the Open Motion Planning Library (OMPL) [41] implementation of RRT in a single-shot approach, OMPL's RRT in an MPC-style approach, MATLAB's `fmincon` in a

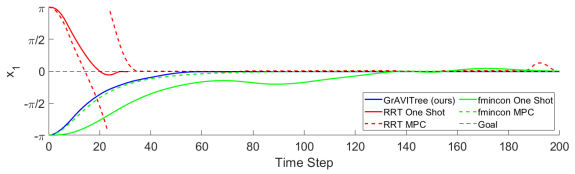


**Fig. 6:** Planned with physics model, executed on physics model. The goal is shown as a thin, dashed black line for reference. GrAVITree stabilized to an error of less than 1 degree from the goal.

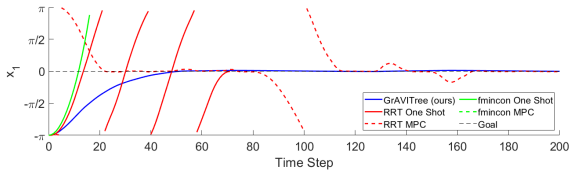


**Fig. 7:** Planned with learned neural network model, executed on neural network model. The goal is shown as a thin, dashed black line for reference. The two `fmincon` trajectories are identical and overlap on the plot. GrAVITree stabilized to within approximately 2 degrees of the goal.

single-shot approach, and MATLAB's `fmincon` in an MPC approach.

Several comments are in order. First, the control scheme for GrAVITree involves derivative-free optimization (15), in which we sample controls to see which gives the lowest expected stage cost plus cost-to-go. Second, the MPC version of RRT involves planning a sequence of states and controls to reach the goal and then executing that plan until the actual state differs from the planned state by more than a small threshold, at which point a new plan is computed. We found that this performed better than executing a fixed number of steps before replanning. Third, the MPC version of the `fmincon` solver adds a terminal state cost equal to 10 times the typical state cost. Without this added penalty at the end of the lookahead, the algorithm did not perform well in any case. Finally, at no point during the third set of experiments was any algorithm given access to the actual physics model for planning.

Figures 6, 7, and 8 show the state trajectories which arose during the experiments. Table I shows the combined cost of the states and controls from each result.

In Fig. 6, we see the results from planning and executing on the physics model. All algorithms achieve some level of success in reaching the goal. GrAVITree, one-shot RRT, and both `fmincon` controllers take fairly direct paths to the goal. However, the one-shot RRT and `fmincon` trajectories have higher costs from faster and slower motion, respectively. The slower one-shot `fmincon` trajectory may constitute a suboptimal local solution. The MPC RRT baseline takes an extra revolution to reach the goal and briefly falls away from the goal near the end but is otherwise successful.

Fig. 7 shows the results from planning and executing on the network model. GrAVITree again takes a direct path to the goal, the one-shot RRT adds multiple oscillations first but

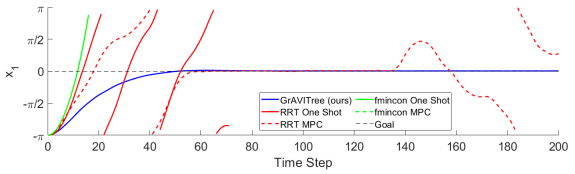


**Fig. 8:** Planned with learned neural network model, executed on physics model. The goal is shown as a thin, dashed black line for reference. The two `fmincon` trajectories are identical and overlap on the plot. Only GrAVITree successfully stabilizes the pendulum under model mismatch. The MPC version of RRT has some success but has trouble stabilizing. GrAVITree stabilized to an error of less than 1 degree from the goal.

| Algorithm | Phys-Phys | Net-Net | Net-Phys |
|---|---|---|---|
| GrAVITree | 187.15 (0.042±0.0015) | 189.65 (0.037±0.0019) | 188.15 (0.037±0.0016) |
| RRT One | 331.28 | 1832.5 | 1652.3 |
| RRT MPC | 1088.4 (0.021±0.033) | 1358.9 (0.031±0.049) | 1838.7 (0.075±0.014) |
| `fmincon` One | 323.9 | Diverged | Diverged |
| `fmincon` MPC | 187.22 (0.055±0.019) | Diverged | Diverged |

**TABLE I:** Trajectory costs. The numbers in parentheses are the mean and standard deviation of computation time in seconds for algorithms that involve recomputation. The GrAVITree times are dominated by the time to evaluate the value function. GrAVITree was run offline for 30 minutes, once for the physics model and once for the learned network model. Note that GrAVITree and RRT were run in C++ but `fmincon` was run in MATLAB, so the times are not directly comparable. All experiments were run on an i9-9900KS CPU.

still reaches the goal, and the MPC RRT is again able to reach the goal but has trouble stabilizing. The main difference here is in the `fmincon` trajectories. Although it is possible to compute a gradient of the network dynamics, curvature is often exceedingly high; hence, gradient-based algorithms such as `fmincon` struggle to cope with the resulting non-convexities. Both reach the plan of using maximum control effort at all times, leading state $x_1$ to perpetually increase (i.e. the pendulum spins around with increasing speed). We cut off the plot after one rotation for clarity.

Fig. 8 shows the results from planning on the learned neural network model and executing on the physics model. This experiment includes the challenges of planning on a network model and dealing with model mismatch. GrAVITree again takes a direct path to the goal. The one-shot RRT, being an open loop plan, fails completely. The MPC RRT is again able to reach the goal but struggles to stabilize the system. Again, both `fmincon` approaches rapidly oscillate and the plots are cut off after one full rotation.

GrAVITree is the only one of these algorithms to successfully reach and stabilize at the goal in all experiments. The MPC `fmincon` approach is very slightly worse on the first experiment, though this difference can be explained by numerical error in planning as small tweaks to parameters led to much greater differences in trajectory costs. Even when treating the MPC RRT as stabilized when first reaching the goal, the trajectory costs for GrAVITree were much lower.

## VIII. Conclusion

We have introduced GrAVITree, a sampling-based algorithm for computing a value function for dynamic systems. The algorithm and its reconfiguration ability was demonstrated on two test systems: a single-integrator and an inverted pendulum. Moreover, our algorithm was used to control a system whose dynamics were modeled by a neural network and successfully handled the resulting model mismatch while remaining robust to gradient irregularities and high curvature in the neural network. Future work will aim to scale GrAVITree to systems with larger state spaces, more complex dynamics, and planning problems in (learned) latent spaces.

## References


[1] Daniel Pfrommer et al. "TaSIL: Taylor Series Imitation Learning". In: *arXiv preprint arXiv:2205.14812* (2022).

[2] Ian M Mitchell, Alexandre M Bayen, and Claire J Tomlin. "A time-dependent Hamilton-Jacobi formulation of reachable sets for continuous dynamic games". In: *IEEE Transactions on automatic control* 50.7 (2005), pp. 947–957.

[3] Somil Bansal et al. "Hamilton-Jacobi reachability: A brief overview and recent advances". In: *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*. IEEE. 2017, pp. 2242–2253.

[4] Alexander B Kurzhanski and Pravin Varaiya. "Ellipsoidal techniques for reachability analysis: internal approximation". In: *Systems & control letters* 41.3 (2000), pp. 201–211.

[5] Alexander B Kurzhanski and Pravin Varaiya. "On ellipsoidal techniques for reachability analysis. part ii: Internal approximations box-valued constraints". In: *Optimization methods and software* 17.2 (2002), pp. 207–237.

[6] Matthias Althoff and Bruce H Krogh. "Zonotope bundles for the efficient computation of reachable sets". In: *2011 50th IEEE conference on decision and control and European control conference*. IEEE. 2011, pp. 6814–6821.

[7] Jianyu Chen, Wei Zhan, and Masayoshi Tomizuka. "Constrained iterative LQR for on-road autonomous driving motion planning". In: *International Conference on Intelligent Transportation Systems (ITSC)*. IEEE. 2017, pp. 1–7.

[8] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[9] Dimitri P Bertsekas. *Dynamic programming and optimal control*. Vol. 1. 2. Athena scientific Belmont, MA, 1995.

[10] Dimitri P Bertsekas. *Approximate dynamic programming*. Athena scientific Belmont, 2012.

[11] Asma Al-Tamimi, Frank L. Lewis, and Murad Abu-Khalaf. "Discrete-Time Nonlinear HJB Solution Using Approximate Dynamic Programming: Convergence Proof". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 38.4 (2008), pp. 943–949. DOI: 10.1109/TSMCB.2008.926614.

[12] Jong-Min Lee and Jay H Lee. "Approximate dynamic programming strategies and their applicability for process control: A review and future directions". In: *International Journal of Control, Automation, and Systems* 2.3 (2004), pp. 263–278.

[13] Rudolf Emil Kalman et al. "Contributions to the theory of optimal control". In: *Bol. soc. mat. mexicana* 5.2 (1960), pp. 102–119.

[14] David Mayne. "A second-order gradient method for determining optimal trajectories of non-linear discrete-time systems". In: *International Journal of Control* 3.1 (1966), pp. 85–95.

[15] Weiwei Li and Emanuel Todorov. "Iterative linear quadratic regulator design for nonlinear biological movement systems." In: *ICINCO*. 2004, pp. 222–229.

[16] Nan Ye et al. "Despot: Online pomdp planning with regularization". In: *Journal of Artificial Intelligence Research* 58 (2017), pp. 231–266.

[17] Zachary N Sunberg and Mykel J Kochenderfer. "Online algorithms for POMDPs with continuous state, action, and observation spaces". In: *Twenty-Eighth International Conference on Automated Planning and Scheduling*. 2018.

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

[19] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[20] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), pp. 484–489.

[21] Shixiang Gu et al. "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates". In: *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2017, pp. 3389–3396.

[22] OpenAI: Marcin Andrychowicz et al. "Learning dexterous in-hand manipulation". In: *The International Journal of Robotics Research* 39.1 (2020), pp. 3–20.

[23] Joshua Achiam et al. "Constrained policy optimization". In: *arXiv preprint arXiv:1705.10528* (2017).

[24] Yongshuai Liu, Jiaxin Ding, and Xin Liu. "IPO: Interior-point policy optimization under constraints". In: *arXiv preprint arXiv:1910.09615* (2019).

[25] Jingqi Li et al. "Augmented Lagrangian method for instantaneously constrained reinforcement learning problems". In: *2021 60th IEEE Conference on Decision and Control (CDC)*. IEEE. 2021, pp. 2982–2989.

[26] Steven M LaValle. "Rapidly-exploring random trees: A new tool for path planning". In: (1998).

[27] Lydia E Kavraki et al. "Probabilistic roadmaps for path planning in high-dimensional configuration spaces". In: *IEEE transactions on Robotics and Automation* 12.4 (1996), pp. 566–580.

[28] Ioan A Şucan and Lydia E Kavraki. "Kinodynamic motion planning by interior-exterior cell exploration". In: *Algorithmic Foundation of Robotics VIII*. Springer, 2009, pp. 449–464.

[29] Sertac Karaman and Emilio Frazzoli. "Sampling-based optimal motion planning for non-holonomic dynamical systems". In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE. 2013, pp. 5041–5047.

[30] Sertac Karaman and Emilio Frazzoli. "Sampling-based algorithms for optimal motion planning". In: *The international journal of robotics research* 30.7 (2011), pp. 846–894.

[31] Jonathan D Gammell, Siddhartha S Srinivasa, and Timothy D Barfoot. "Batch informed trees (BIT*): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs". In: *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2015, pp. 3067–3074.

[32] James J Kuffner and Steven M LaValle. "RRT-connect: An efficient approach to single-query path planning". In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*. Vol. 2. IEEE. 2000, pp. 995–1001.

[33] Jeffrey Ichnowski and Ron Alterovitz. "Motion planning templates: A motion planning framework for robots with low-power CPUs". In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 612–618.

[34] Ryan Luna et al. "A scalable motion planner for high-dimensional kinematic systems". In: *The International Journal of Robotics Research* 39.4 (2020), pp. 361–388.

[35] Lucas Janson et al. "Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions". In: *The International journal of robotics research* 34.7 (2015), pp. 883–921.

[36] Ross Allen and Marco Pavone. "A real-time framework for kinodynamic planning with application to quadrotor obstacle avoidance". In: *AIAA Guidance, Navigation, and Control Conference*. 2016, p. 1374.

[37] Russ Tedrake et al. "LQR-trees: Feedback motion planning via sums-of-squares verification". In: *The International Journal of Robotics Research* 29.8 (2010), pp. 1038–1052.

[38] Florent Lamiraux, Etienne Ferré, and Erwan Vallée. "Kinodynamic motion planning: Connecting exploration trees using trajectory optimization methods". In: *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*. Vol. 4. IEEE. 2004, pp. 3987–3992.

[39] Robert Hooke and T. A. Jeeves. ""' Direct Search" Solution of Numerical and Statistical Problems". In: *J. ACM* 8.2 (Apr. 1961), pp. 212–229. ISSN: 0004-5411. DOI: 10.1145/321062.321069. URL: https://doi.org/10.1145/321062.321069.

[40] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010.

[41] Ioan A Şucan, Mark Moll, and Lydia E Kavraki. "The open motion planning library". In: *IEEE Robotics & Automation Magazine* 19.4 (2012), pp. 72–82.