Near-Optimal Search in Continuous Domains

Samuel Ieong*, **Nicolas Lambert** and **Yoav Shoham**
Computer Science Department
Stanford University
{sieong,nlambert,shoham}@cs.stanford.edu

Ronen Brafman
Computer Science Department
Ben Gurion University
brafman@cs.bgu.ac.il

Abstract

We investigate search problems in continuous state and action spaces with no uncertainty. Actions have costs and can only be taken at discrete time steps (unlike the case with continuous control). Given an admissible heuristic function and a starting state, the objective is to find a minimum-cost plan that reaches a goal state. As the continuous domain does not allow the tight optimality results that are possible in the discrete case (for example by A*), we instead propose and analyze an approximate forward-search algorithm that has the following provable properties. Given a desired accuracy ϵ, and a bound d on the length of the plan, the algorithm computes a lower bound L on the cost of any plan. It either (a) returns a plan of cost L that is at most ϵ more than the optimal plan, or (b) if, according to the heuristic estimate, there may exist a plan of cost L of length $> d$, returns a partial plan that traces the first d steps of such plan. To our knowledge, this is the first algorithm that provides optimality guarantees in continuous domains with discrete control and without uncertainty.

Introduction

From path planning in the Mars Rover (Bresina *et al.* 2002; Mausam *et al.* 2005) to flying autonomous helicopters (Abbeel *et al.* 2007), planning problems in practice often have to deal with continuous parameters. In this paper, we study planning problems that feature continuous state and action spaces with discrete control, i.e., actions can only be taken at discrete time steps. We start by giving two examples where we may come across such problems.

Imagine we are asked to park a Mars Rover through remote controls. As communication consumes energy, and can take place only over certain windows of opportunities (when the Rover is on the correct side of Mars), it is not possible to have continuous control over the Rover's actions. To park the Rover, we may choose to send a few discrete commands, such as "turn the steering wheel 15 degrees to the left and go in arc for 2 meters." Note that while control is discrete, both the state space and the action space are continuous.

Closer to home, consider a planning problem from the computer billiards championships (Smith 2006). There, contesting algorithms are asked to submit a cue action from a six-dimensional continuous action space. Once this action is selected, the shot is simulated and the result is fed back to the contestant. Note that once the action is selected, there is no continuous control over the cue ball, as the ball will roll according to the laws of physics based on the initial action.

Despite much work on planning from both AI and motion planning research, little is known about whether such problems can be solved with strong theoretical guarantees. Our main contribution is to present a novel forward-search algorithm that, for a given desired accuracy ϵ and a bound d on the length of the plan, computes a lower bound L on the cost of any plan. It returns either a complete plan that is at most ϵ worse than the best possible one, or a partial plan for which, according to the heuristic estimate, there may exist a complete plan of cost L of length $> d$ that uses the partial plan. This is guaranteed under mild assumptions. The main idea is to explore the action space in a disciplined manner without arbitrary discretization, and through the use of any admissible heuristic function, prune away poor policies and focus the search on more promising directions.

Given the vast amount of research done on planning in continuous domain, it is impossible to thoroughly discuss the applicability of the many different algorithms to our problem domain. At a high level, we can identify techniques from two main sources: MDPs and motion planning.

Many continuous planning problems have been studied under the framework of MDPs, POMDPs (Roy & Thrun 2002) and control theory. This problem is different from ours, as their solutions use backward induction to deal with uncertainties, while without uncertainty the use of forward, heuristic search allows more efficient computations. Besides, while there are good theoretical techniques for dealing with continuous state spaces (Hauskrecht & Kveton 2004; Guestrin, Hauskrecht, & Kveton 2004), relatively little work has explicitly dealt with continuous action spaces. When faced with such problems, either *ad hoc* discretization of the action space or sampling is used (Porta *et al.* 2006).

Unlike MDPs, motion planning algorithms often work directly with continuous state and action spaces (Latombe 1991). In particular, sampling-based algorithms, such as probabilistic roadmaps (PRMs) (Kavraki *et al.* 1996) and rapidly exploring random trees (RRTs) (LaValle & Kuffner 1999), do not require discretization of the state or action

*Supported by supported by a Richard and Naomi Horowitz Stanford Graduate Fellowship.

space. However, their goal is to decide if a path exists between two configurations (a start state and a goal state); the optimality of the path is not dealt with explicitly. More importantly, most motion planning algorithms assume continuous control. When decisions occur in discrete time, the constraints are non-differentiable (LaValle 2006). There exist few algorithms for motion planning under these constraints, and little is known about their theoretical guarantees.

The paper is organized as follows. We start by giving a formal definition of the planning problem in consideration and the underlying assumptions. Then we propose an algorithm that solves such problems, and provide theoretical guarantees. We conclude with some possible extensions to our work presented here.

Model & Terminology

We start by defining planning problems in continuous state and action spaces with discrete control.

Definition 1. An instance of a planning problem in our domain is given by $\langle \mathcal{S}, \mathcal{A}, \mathcal{G}, T, C, H, s_0 \rangle$, where

- $\mathcal{S} \subseteq \mathbb{R}^n$ is an n-dimensional continuous *state space*
- $\mathcal{A} = [\underline{a}, \overline{a}] \subset \mathbb{R}^m$ is an m-dimensional continuous *action space*, given by a hyper-rectangle, bounded by the two vectors $\underline{a}$ and $\overline{a}$, which gives the minimum and maximum controls in the m dimensions
- $\mathcal{G} \subset \mathcal{S}$ is the set of goal states (*goal set*)
- $T : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{S}$ is the *transition function* that maps a given state and an action to its destination state
- $C : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}_+$ is the *cost function* that returns a non-negative cost of the action taken at a given state
- $H : \mathcal{S} \mapsto \mathbb{R}_+$ is an *admissible heuristic function* that returns a lower bound (under-estimate) on the cost of any plan from a given state to any goal state
- $s_0 \in \mathcal{S}$ is the *starting state* $\triangleleft$

We refer to a sequence of actions as a *plan*. A plan is *complete* if the sequence of actions, $(a_0, a_1, \ldots, a_k)$ leads to a goal state, i.e.,

$$T(\cdots T(T(s_0, a_0), a_1), \ldots, a_k) \in \mathcal{G}$$

otherwise the plan is *partial*. The cost of a plan is given by

$$\sum_{i=0}^{k} C(s_i, a_i)$$

where s_i is the state before the i-th action takes place.

We assume the planning problems to satisfy two assumptions. The first ensures the existence of a complete plan for the problem.

Assumption 1. The goal set $\mathcal{G}$ is open, and there exists at least one complete plan. $\triangleleft$

Our next assumption has to do with the continuity of the transition, the cost, and the heuristic functions. We first define *Lipschitz continuity*.

Definition 2. A function $f : \mathbb{R}^p \mapsto \mathbb{R}^q$ is *Lipschitz-continuous with constant K* if for all $a, b \in \mathbb{R}^p$,

$$||f(a) - f(b)|| \leq K||a - b||$$

$||\cdot||$ may be any norm, for instance the Euclidean norm. $\triangleleft$

Assumption 2 (Lipschitz continuity)**.** The transition function $T(s, a)$, cost function $C(s, a)$, and heuristic function $H(s)$, are Lipschitz-continuous. For the heuristic function, we denote its Lipschitz constant by h_s. For the transition and the cost functions, we denote their constants by t_s, t_a, c_s, c_a. Note that there are two constants per function, one for state (s) and one for action (a). For the transition function $T(\cdot, \cdot)$,

$$||T(s, a) - T(s', a)|| \leq t_s||s - s'||$$
$$||T(s, a) - T(s, a')|| \leq t_a||a - a'||$$

and similarly for $C(\cdot, \cdot)$. This finer distinction follows from the basic definition of Lipschitz continuity. $\triangleleft$

Algorithm

We now present our approximate forward-search algorithm that solves the planning problem previously stated[1].

Data Structure

The basic idea of the forward-search algorithm is to build a search tree in the continuous domain. At each node of the tree, we store the following data:

- $state$: the state s, associated to the node
- $action$: the action a that leads from the parent state to s
- $depth$: the depth of the node in the search tree
- $lastCost$: the cost of the action a that leads to this state
- $estimate$: a lower bound on the minimum cost of any plan from this state to a goal state
- M: a *saw-tooth* function, such that for any action α, it returns the *lower bound* on the estimated cost of the plan that takes action α from state s.
- $parent$: the parent node
- $children$: the set of children

Before we proceed to the description of the algorithm, we provide some insight on the use of the function M, as it constitutes the soul and backbone of our algorithm. Recall that by Assumption 2, the transition function, the cost function, and the heuristic function are all Lipschitz continuous. Thus, at a given state s, the heuristic estimate of $H(T(s, a))$ provides a bound on the heuristic estimate on actions other than a. In particular, for any action a',

$$\begin{aligned} ||H(T(s, a')) - H(T(s, a))|| &\leq h_s||T(s, a) - T(s, a')|| \\ &\leq h_s t_a||a - a'|| \end{aligned}$$

Similarly, for the cost function,

$$||C(s, a') - C(s, a)|| \leq c_a||a - a'||$$

[1] Our topological assumption on goal sets implies that, in general, there is no optimal plan, as by continuity of costs and transitions, complete plans may be improved by infinitesimal modifications. The optimal cost is therefore defined as the infimum over all sequences of complete plans. We nevertheless use the term *optimal plan* for simplicity.

Thus, the minimum cost of a plan that takes action a' from state s, $M(a')$, is bounded by

$$\begin{aligned} M(a') &= C(s,a') + H(T(s,a')) \\ &\geq C(s,a) + H(T(s,a)) - (h_s t_a + c_a)||a - a'|| \\ &= M(a) - (h_s t_a + c_a)||a - a'|| \end{aligned}$$

The saw-tooth function M helps us to keep track of these cost lower bounds, as illustrated by Figure 1. In the figure, only actions a_1 and a_2 have been sampled. The lower bound on the rest of the actions is given by the bounding technique described.

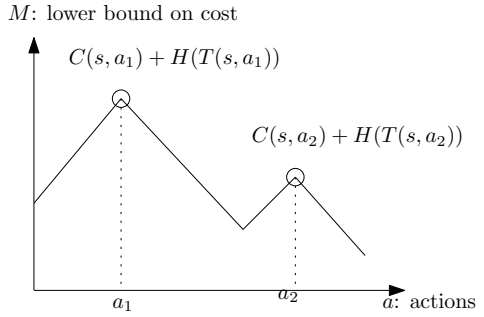

Figure 1: Saw-tooth function M

Base algorithm

The basic idea of the algorithm is to construct a search tree in the continuous domain. At each node, the algorithm keeps track and updates the saw-tooth function M based on information obtained by sampling new actions. It is structured as follows:

Main Algorithm Continuous Search $(\epsilon, MaxDepth)$

```
root ← single leaf node with state s_0, and estimate H(s_0)
loop
    ⟨ 1. find the node with lowest estimated cost ⟩
    if node.state ∈ G or node.depth = MaxDepth then
        return root.estimate and path from root to node
    end if
    if node is a leaf then
        ⟨ 2. expand node ⟩
    else
        ⟨ 3. resample node ⟩
    end if
    ⟨ 4. update the tree ⟩
end loop
```

The algorithm starts by setting the root of the search tree to be the starting state s_0 of the planning problem. At each step of the loop, it finds the node in the tree that has the lowest estimated cost. This is performed at Step 1:

⟨ 1. find the node with lowest estimated cost ⟩

```
node ← root
previous ← null
while previous ≠ node and node is not a leaf do
    previous ← node
    child ← arg min_{c∈node.children} c.estimate + c.lastCost
    bestEstimate ← child.estimate + child.lastCost
    if bestEstimate − node.estimate ≤ ε/2^{node.depth} then
        node ← child
    end if
end while
```

Once this node is found, if it is a goal state, then the algorithm has found a plan that is guaranteed to be within ϵ of the cost of the optimal plan. If $MaxDepth$ is reached, then the algorithm will return a partial plan that ends in a node at $MaxDepth$ with the lowest estimated cost to goal. Otherwise, the algorithm will sample some actions. If the node is visited for the first time (thus a leaf), it will sample the "average" action, $(\underline{a} + \overline{a})/2$. There are other possible choices, this choice is mostly for concreteness of our description.

⟨ 2. expand a leaf $node$ ⟩

```
child ← Create node corresponding to (a_ + a¯)/2
Update node.M using constant (h_s t_a + c_a)
node.estimate ← min_a node.M(a)
```

When a new node is created, its cost estimate is provided by the heuristic function. The last two steps require additional clarification. As mentioned earlier, the cost and heuristic value of sampled actions can be used to bound estimated costs of unseen samples. Therefore, after sampling actions at a node, the function M needs to be updated. Let M' be the saw-tooth function after the update. Its values are determined by the point-wise maximum of the original function M and the new "cone" due to the sample. Suppose a is the new sample, then $\forall \alpha \in \mathcal{A}$,

$$M'(\alpha) = \max(M(\alpha), C(s,a) + H(s,a) - (h_s t_a + c_a)||\alpha - a||)$$

As we will see shortly, the Lipschitz constants for the updates may differ. After M is updated, we set the new estimate of the node according to the minimum in M.

When a node has already been expanded, but no previously sampled action may lead to the estimated optimal cost, new actions are sampled. The action to sample next depends on which action shows the most promise (i.e., a minimum) according to the function M. This is performed in Step 3:

⟨ 3. resample non-leaf $node$ ⟩

```
a* ← arg min_a node.M(a)
child ← Create node according to a*
Update node.M using constant (h_s t_a + c_a)
node.estimate ← min_a node.M(a)
```

Whether the algorithm has expanded a new node, or resampled at an existing node, if the estimate of the node has changed, it needs to check with its parent to see if this may in turn change the estimate of its parent. These updates may propagate upwards until the root node, or until the estimate of some ancestor of the node remains unchanged. The formula used as Lipschitz constant for updating the function M will be explained in the theoretical analysis.

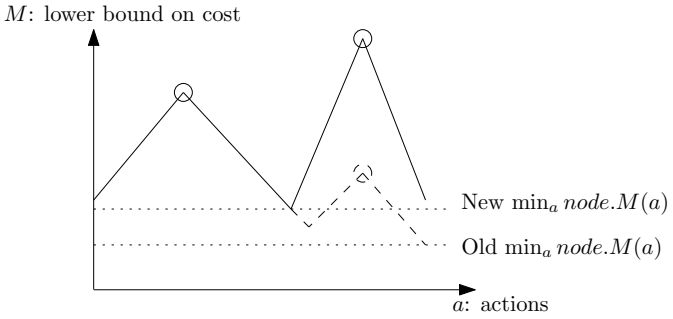

Figure 2: Updating Saw-tooth function M

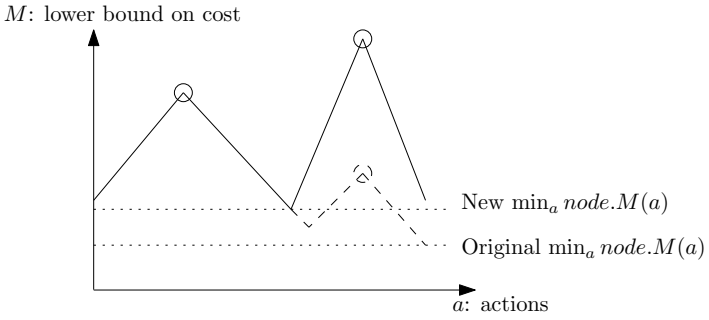

Figure 3: New lower bound discovered

⟨ 4. update the tree ⟩

$d \leftarrow 1$
while $node$ has a parent and $node.estimate$ has changed **do**
  $d \leftarrow d + 1$
  $node \leftarrow node.parent$
  $newLipschitz \leftarrow c_a + (\sum_{i=1}^{d-1} c_s t_a t_s^{i-1}) + h_s t_a t_s^{d-1}$
  Update $node.M$ using constant $newLipschitz$
  $node.estimate \leftarrow \min_a node.M(a)$
  for each $c \in node.children$ **do** update $c.estimate$
end while

The situation in step 4 of the algorithm can be summarized in figures 2 and 3. In Figure 2, suppose we have expanded the leaf node at a_2, and discover a new lower bound on the cost of any plan that takes action a_2, we will then update the function M at node s. Note that this update may involve a larger Lipschitz constant, and hence the slope of the update may be steeper. After the update, we may discover a new lower bound on the cost of any plan starting from node s, as illustrated in Figure 3. This in turn may trigger further updates upstream in the tree.

Handling updates of function M

So far we abstracted away the internal representation and the operations on M. However in action spaces of dimension 2 and higher, the saw-tooth function M is an *envelope of cones* oriented downwards, and finding its minimum is known to be a hard problem (Mladineo 1986). Therefore, instead of working directly on the cones, we use an *envelope of hyper-rectangles*: the function M is approximated as a piecewise-constant function $\tilde{M}$ taking constant values on subintervals that partition the action space. By ensuring that $\tilde{M}$ is always lower than M, its minimum remains a lower-bound of the cost of the optimal plan. Instead of using M's minimum, we estimate the cost of the optimal plan by the value of the lowest hyper-rectangle.

In the $node$ data structure, M is replaced by a collection of $cones$ and a collection of hyper-rectangles $rects$. Cones can be represented as tuples $(value, action, \lambda)$, and define the function $x \mapsto value - \lambda||action - x||$. Hyper-rectangles are tuples $([a, b], value)$, and define a hyper-rectangle surface of dimensionality of the action space, with height $value$.

The approximate $\tilde{M}$ is kept as close to M as possible by raising the rectangle envelope whenever the partition is refined or M is updated. These raising operations are performed by calls to the UPDATE function, which places the hyper-rectangles located below a cone at the highest position beneath it.

⟨ Update a hyper-rectangle's value ⟩

function UPDATE($rectangle, cone$)
  Let $([a, b], v) = rectangle$
  $x \leftarrow$ farthest vertex of the multidimensional interval $[a, b]$ from $cone.action$
  $rectangle.value \leftarrow \max(rectangle.value, cone.value - cone.\lambda * ||x - cone.action||)$
end function

The change to envelope of rectangles requires some changes to the steps. Node expansion is now done by creating two children corresponding to the boundary actions, $\underline{a}$, $\overline{a}$, and their associated cones. One hyper-rectangle covering the entire action space is created for the newly expanded node, with its value set to be as high as possible while remaining below the envelope formed by the two cones. The new estimate (of the cost of the optimal plan) at the new node is given by the hyper-rectangle's value.

⟨ 2. expand a leaf $node$ ⟩

Create $child_1$ and $child_2$ corresponding to actions $\underline{a}$ and $\overline{a}$
$node.children \leftarrow node.children \cup \{child_1, child_2\}$
$cone_1 \leftarrow (a, child_1.estimate + child_1.lastCost, c_a + h_s t_a)$
$cone_2 \leftarrow (b, child_2.estimate + child_2.lastCost, c_a + h_s t_a)$
$node.cones \leftarrow node.cones \cup \{cone_1, cone_2\}$
$R \leftarrow ([\underline{a}, \overline{a}], \ldots$
$\ldots \min_{c \in \{child_1, child_2\}} c.lastCost + c.estimate \ldots$
$\ldots - (c_a + h_s t_a)||child_1.action - child_2.action||$
$node.rects \leftarrow node.rects \cup \{R\}$
$node.estimate \leftarrow R.value$

As in the base algorithm, resampling occurs when none of the current samples would allow to reach the current estimate of the optimal cost, here provided by the hyper-rectangle of lowest value R. Resampling takes two steps. We first split R in two by dividing its corresponding multi-dimensional interval of actions $[a_1, b_1]$ along the longest edge into two subintervals of equal size, $[a_1, b_2]$ and $[a_2, b_1]$. This refines $\tilde{M}$'s partition of actions. The envelope of hyper-rectangles is then raised to be as close to the lower-bound of the cone envelope as possible. We then sample the actions a_2

and b_2, and create the associated children and cones. As we modify the cone envelope, we raise the collection of hyperrectangles one more time to tighten the approximation.

⟨ 3. resample non-leaf $node$ ⟩

$R \leftarrow \min_{R \in node.rects} R.value$
Let $([a_1, b_1], v) = R$: divide $[a_1, b_1]$ along the longest edge into two intervals of equal size,$[a_1, b_2]$ and $[a_2, b_1]$
$R_1 \leftarrow ([a_1, b_2], v)$ and $R_2 \leftarrow ([a_2, b_1], v)$
for each $cone \in node.cones$ **do** UPDATE($R_1, cone$) . . .
. . . and UPDATE($R_2, cone$)
$node.rects \leftarrow node.rects \cup \{R_1, R_2\} \backslash \{R\}$
Create $child_1$ and $child_2$, corresponding to actions a_2 and b_2
$node.children \leftarrow node.children \cup \{child_1, child_2\}$
$cone_1 \leftarrow (child_1.estimate, a_2, c_a + h_s t_a)$
$cone_2 \leftarrow (child_2.estimate, b_2, c_a + h_s t_a)$
$node.cones \leftarrow node.cones \cup \{cone_1, cone_2\}$
for each $R \in node.rects$ **do** UPDATE($R, cone_1$) . . .
. . . and UPDATE($R, cone_2$)
$node.estimate \leftarrow \min_{R \in node.rects} R.value$

Finally the tree is updated upwards from the current node to the root. For each node encountered, a new cone is added to the node's parent, corresponding to the action taken from the parent to the node. The slope coefficient is the same as in the base algorithm. We again raise the envelope of hyperrectangles $\tilde{M}$ to lie just below M.

⟨ 4. update the tree ⟩

$d \leftarrow 1$
while $node$ has a parent and $node.estimate$ has been changed **do**
$d \leftarrow d + 1$
$cone \leftarrow (node.action, node.estimate, c_a + (\sum_{i=1}^{d-1} c_s t_s^{i-1} t_a) + h_s t_s^{d-1} t_a)$
$node \leftarrow node.parent$
$node.cones \leftarrow node.cones \cup \{cone\}$
for each $R \in node.rects$ **do** UPDATE($R, cone$)
$node.estimate \leftarrow \min_{R \in node.rects} R.value$
end while

Formal guarantees

We now describe the formal guarantees of our algorithm, listed below:

Theorem 1. *Main Algorithm terminates after a finite number of steps.*

Theorem 2. *When the algorithm terminates, it returns a lower bound on the cost of any plan.*

Theorem 3. *If the algorithm returns a complete plan, then its cost is at most ϵ more than that of the optimal plan.*

Theorem 4. *If the algorithm returns a partial plan, then there may exist a plan of cost at most ϵ more than that of the optimal plan that uses the partial plan.*

To clarify Theorem 4, when Main Algorithm returns a partial plan, that is because the algorithm is going to next resample a node s at maximum depth. This happens only when the sum of the cost of the partial plan and the heuristic estimate at s is at most ϵ higher than the lower bound on the cost of any plan. While this does not guarantee the existence of an ϵ-optimal plan that goes through s, if the heuristic estimate is reliable, there is likely to be an ϵ-optimal plan that can be derived from the partial plan.

To prove these theorems, we first need to establish a couple of lemmas. The first bounds the maximum difference in costs of plans that take different actions from the same state.

Lemma 1. *Let $L_k(s, a)$ denote the lower bound on the cost of any plan that starts in state s, take action a, and continues for k more steps, defined by*

$$L_k(s,a) = C(s,a) + \min_{a_1, a_2, \ldots, a_k} \left(\sum_{i=1}^{k} C(s_i, a_i) + H(s_{k+1}) \right)$$

where $s_i = T(s_{i-1}, a_{i-1})$. For a different action $a' \neq a$,

$$||L_k(s,a') - L_k(s,a)|| \leq N(k)||a' - a||$$

where $N(k) = c_a + \left(\sum_{i=0}^{k-1} c_s t_a t_s^i\right) + h_s t_a t_s^k$.

Note that this is a generalization of the relationship we mentioned in the data structure section when we describe the data structure M. In particular, that relationship is a special case of Lemma 1 for $k = 0$. This also explains the constant we use in Step 4 of the algorithm. The proof is technical and can be found in the Appendix.

Our second lemma establishes an invariance that is maintained by the main loop of the algorithm.

Lemma 2. *At the beginning of each loop, $root.estimate$ is a valid lower bound on the cost of the optimal plan.*

Proof. We will prove this by induction. When we initialize the algorithm, $root.estimate$ is set to be $H(s_0)$. By admissibility of the heuristic function, this lower bound is valid.

Suppose $root.estimate$ is correct after the n-th loop. In the next loop, $root.estimate$ may be changed in two places. It may be changed either (1) when we sample actions from the root, or (2) in Step 4 ⟨ update the tree ⟩.

Case (1): This is an application of Lemma 1 for $k = 0$.

Case (2): in Step 4, $root.estimate$ may be updated when the estimate of some leaf node at depth d is changed. This may happen when the change causes cascading updates that propagate up the tree. Lemma 1 for $k = d$ applies if $root.estimate$ gets changed. □

Proof (Theorem 2). This is a consequence of Lemma 2 when the algorithm terminates. □

Proof (Theorem 3). In Step 2 of the algorithm, the node chosen to be expanded or resampled is at most ϵ more than $root.estimate$. By Lemma 2, since that is a lower bound on the cost of any plan, if the node chosen is in a goal state, the plan costs at most ϵ more than that of the optimal plan. □

Proof Sketch (Theorem 4). This follows from similar reasons in the proof of Theorem 3 and is omitted. □

We still need to establish that the algorithm terminates after a finite number of steps. Intuitively, this is not obvious because there are uncountably many actions in the search space. However, the search space can be pruned using Lemma 1. The complete proof is as follows.

Proof (Theorem 1). Suppose the algorithm does not terminate for a given problem. This is only possible if for all the nodes examined (i.e, expanded or resampled), none of them is in the goal set nor at maximum depth. If a node is examined, and it does not cause an update, then the next node to be examined will be a child of the node (by invariance of Lemma 2 and Step 2). Hence, for the algorithm not to terminate, updates must take place.

For any given node, since the Lipschitz constants for the update is bounded, after a certain number of its children has been examined, its estimate will have to increase. Repeating this argument, eventually, the estimate of the root node has to increase. In particular, if the algorithm does not terminate, the estimate of the root will have to increase unboundedly.

However, by Assumption 1, there exists a complete plan for the planning problem. This plan has finite cost. Therefore, $root.estimate$ cannot be unboundedly large, since by Lemma 2, it is always a lower bound on the cost of any plan. This contradicts the previous claim that estimate at root continues to increase unboundedly. □

Concluding Remarks

In this paper, we looked at planning problems that feature continuous state and action spaces with discrete control. We formulated the problem as a heuristic search problem, and proposed an approximate forward-search algorithm with concrete theoretical guarantees. The key algorithmic insight is to make use of the Lipschitz continuity in the transition, the cost, and the heuristic functions to intelligently prune away bad plans. While we have explained this technique in the context of a specific algorithm, the idea may also be applicable to other methods, such as grid-based techniques that uniformly discretize the action space.

Although we provided precise suggestions as to how the algorithm may be implemented, our main contribution remains theoretical. Experiments with a rudimentary implementation show that, at comparable tree sizes, our algorithm has similar performances as grid-based methods (but such methods do not offer control for error bounds). However, we believe that further optimization will lead to significant improvements, and we hope to focus our next contribution on practical aspects of our algorithm.

Finally, one important direction we plan to explore is to extend our method to cater for uncertainties in action outcomes. This may provide an alternative solution technique to continuous MDPs by means of forward search.

Acknowledgments

We thank Jean-Claude Latombe and Andrew Ng for fruitful discussions on related algorithms for the problem.

References

Abbeel, P.; Coates, A.; Quigley, M.; and Ng, A. Y. 2007. An application of reinforcement learning to aerobatic helicopter flight.

Bresina, J.; Dearden, R.; Meuleau, N.; Ramkrishnan, S.; Smith, D.; and Washington, R. 2002. Planning under continuous time and resource uncertainty: A challenge for AI. In *Proc. 18th Conf. on Uncertainty in Artificial Intelligence*, 77–84.

Guestrin, C.; Hauskrecht, M.; and Kveton, B. 2004. Solving factored mdps with continuous and discrete variables.

Hauskrecht, M., and Kveton, B. 2004. Linear program approximations for factored continuous-state markov decision processes. In *NIPS 16*, 859–902.

Kavraki, L. E.; Svestka, P.; Latombe, J.-C.; ; and Overmars, M. 1996. Probabilistic roadmaps for path planning in high dimensional configuration spaces. In *IEEE Trans on Robotics and Automation*, volume 12, 566–580.

Latombe, J.-C. 1991. *Robot Motion Planning*. Kluwer Academic Publishers.

LaValle, S. M., and Kuffner, J. J. 1999. Randomized kinodynamic planning. In *Proc. IEEE Int. Conf. Robotics and Automation*, 473–479.

LaValle, S. M. 2006. *Planning Algorithms*. Cambridge University Press.

Mausam; Benazera, E.; Brafman, Ronen Meuleau, N.; and Hansen, E. A. 2005. Planning with continuous resources in stochastic domains. In *Proc. 19th Int. Joint Conf. on Artificial Intelligence*, 1244–1251.

Mladineo, F. H. 1986. An algorithm for finding the global maximum of a multimodal, multivariate function. *Math. Program.* 34(2):188–200.

Porta, J. M.; Vlassis, N.; Spaan, M. T.; and Poupart, P. 2006. Point-based value iteration for continuous POMDPs. *J Machine Learning Research* 7:2329–2367.

Roy, N., and Thrun, S. 2002. Motion planning through policy search. In *Proc. IEEE Int. Conf. Intell. Robots and Sys.*

Smith, M. 2006. Running the table: An AI for computer billiards. In *Proc. 21st Nat. Conf. on Artificial Intelligence*.

Appendix

To prove Lemma 1, we use the following inequality.

Lemma 3. *For two functions over the same domain, $f(\cdot)$ and $g(\cdot)$,*

$$|| \min_x f(x) - \min_y g(y)|| \leq \max_z ||f(z) - g(z)||$$

Proof Sketch (Lemma 1). We establish the case for $k = 1$ to build intuition. The general case can be proved inductively. By definition,

$$L_1(s,a) = C(s,a) + \min_{a_1} \Big(C(s_1,a_1) + H(s_2) \Big)$$
$$L_1(s,a') = C(s,a') + \min_{a'_1} \Big(C(s'_1,a'_1) + H(s'_2) \Big)$$

By triangular inequality and Lemma 3,

$$\begin{aligned} &||L(s,a') - L(s,a)|| \\ &\leq ||C(s,a') - C(s,a)|| && (1) \\ &+ \max_x ||C(T(s,a'),x) - C(T(s,a),x)|| && (2) \\ &+ \max_y ||H(T(T(s,a'),y)) - H(T(T(s,a),y))|| && (3) \end{aligned}$$

We can bound the terms in Equations (1), (2), and (3) by repeated application of the Lipschitz condition. Hence,

$$||L_1(s,a') - L_1(s,a)|| \leq (c_a + c_s t_a + h_s t_s t_a)||a' - a||$$

□