Feedback-based Continuous planning for G-TAMP

Beomjoon Kim

December 16, 2018

1 Problem formulation

Given a computational budget B, a plan skeleton $\{\mathfrak{o}_1(\delta_1,\cdot),\cdots,\mathfrak{o}_T(\delta_t,\cdot)\}$, and the set of goal states S_G, possibly described with a predicate, find the continuous parameters $\kappa_1,\cdots,\kappa_T$ such that it maximizes the sum of the discounted rewards,

$$\max_{\kappa_1,\cdots,\kappa_T}\sum_{t=0}^{T}\gamma^t r(s_t,\kappa_t)$$

where $r(s_t,\kappa_t)=r(s_t,\mathfrak{o}_t(\delta_t,\kappa_t))$, $s_T\in S_G$ and the generative model of the environment T such that $s_{t+1}=T(s_t,\mathfrak{o}_t(\delta_t,\kappa_T))$

2 Voronoi-based Optimistic Optimization for continuous space

Suppose for now that the planning horizon is 1. This reduces to the black-box function optimization problem.

We would like to find the optimal action x^* under the assumption that

$$f(x)-f(y)\le\lambda\cdot d(x,y)$$

So, we have $f(x)\le f(y)+\lambda\cdot d(x,y)$.

At any time point t, we know that we would have evaluated $x_1,\cdots x_t$ number of points. We denote the Vornoi regions induced by these points as $V(x_i)$. By the definition of the Voronoi region, we know that the lowest upper-bound of value of points in a region is provided by the furthest distance to the generator x_i of the region. Hence, we propose the following:

1. Compute the convex hulls of the Voronoi regions $V(x_1),\cdots,V(x_t)$. Denote $CH(V_i)$ as the convex hull of the i^{th} Voronoi region.

2. Select the most optimistic region by

$$i^*=\arg\max_{i\in 1,\cdots,t}f(x_i)+\lambda\cdot\max_{x\in CH(V_i)}d(x,x_i)$$

3. Randomly sample a point from $V(x_{i^*})$.

2.1 Planning with VOO

2.1.1 Monte-Carlo planning with progressive widening

Algorithm 1 SEARCH(s_0)

```
1: $\mathcal{T}(s_0) = \{N(s_0) = 1, A = \emptyset, Q(s_0, \cdot) = 0, N(s_0, \cdot) = 0\}$
2: **repeat**
3:     SIMULATE($s_0, 0$)
4: **until** *timeout*
5: **return** $\arg\max_{a \in \mathcal{T}(s_0).A} \mathcal{T}(s_0).Q(s, a)$
```

Algorithm 2 SIMULATE($s, T, \delta, \alpha, \epsilon, S_G$)

```
1: **if** $s == infeasible$ **then**
2:     **return** infeasible-rwd
3: **else if** $s \in S_G$ **then**
4:     **return** success-rwd
5: **end if**
6: **if** $|\mathcal{T}(s).A| \leq \delta \cdot \mathcal{T}(s).N(s)$ **then**
7:     //progressive widening step
8:     $a \sim$VOO($\mathcal{T}(s).A, \epsilon$)// based on the algorithm above
9:     $\mathcal{T}(s).A = \mathcal{T}(s).A \cup a$
10: **else**
11:     $a = \arg\max_{a \in tree(s).A} Q(s, a, \alpha)$
12: **end if**
13: $s', r \sim f(s, a)$
14: $R = r + \gamma \cdot$SIMULATE($s', depth + 1$)
15: $\mathcal{T}(s).N(s, a) += 1$
16: $\mathcal{T}(s).Q(s, a) = Q(s, a) + \frac{R - \mathcal{T}(s).Q(s,a)}{\mathcal{T}(s).N(s,a)}$
```

Here, $\mathcal{T}$ refers to the Monte Carlo tree, $N(s)$ is the number of times a state s has been visited, $N(s, a)$ is the number of times a has been simulated from s, and $Q(s, a)$ is the value of the action edge a that goes out from s. Q^+ is the augmented Q function that encourages exploration. π is the stochastic policy which we sample actions from, and δ is the progressive widening parameter.

This Monte-Carlo planning approach is more advantageous than heuristic-based search in that it is anytime, and the value of the heuristic function improves overtime. It begins with a rather wrong estimate of the value function, 0, and the value becomes more accurate as more simulations are performed.

There are subtleties that need to be taken care of when applying this for TAMP problems. First, instead of depth, we should think about detecting the dead-ends. However, detecting this is non-trivial. Second, the function SAMPLEACTION in line 10 of Algorithm 2 must take account of the fact that there are infeasible actions. Unlike the game of Go where the infeasible moves are cheap and trivial to detect, in TAMP, we need to call a motion planner to find if the action is actually achievable. This is an expensive step.

2.1.2 A variant of A*

The above approach is troublesome in that the function Q is being constantly updated, and that at a particular node, even if you choose the same action you end up with different Q-value. Also, it has too many parameters

We might naively try to optimize the entire sequence $\kappa_1, \cdots, \kappa_T$ directly, but doing so discards the important advantage of the search-based method, which is that it has the ability to back-track, if deemed necessary. We take the idea from A* to optimize $K^T = \kappa_1, \cdots \kappa_T$.

We will denote the i^{th} sequence of actions whose length is $d \in [1, T]$ as $K_i^d = \kappa_1^{(i)} \cdots \kappa_d^{(i)}$. Further, we assume that the reward function is Lipschitz.

1. Inputs: VOO parameter ϵ, heuristic function $\hat{h}(K^*)$
2. Initialize the sequence set S with an empty set.
3. for i=1 to timeout,
4. Choose a sequence $K^d \in S$ according to

$$B(K^d) = \sum_{t=1}^{d-1} \gamma^t r(s_t, \kappa_t) + diameter(V(s_t.K)) + \gamma^d \hat{h}(K^d)$$

5. Choose the dimension to cut, from $d+1$ choices, that would reduce the upper bound $B(K^d)$ the most.
6. Sample an action $a \sim$VOO(ϵ), and divide the selected dimension
7. Add the new sequence to the sequence set S

Alternatively, what happens if we ignore the diameter information? It will not take account of how many actions we've s sampled at a particular dimension in K^T.

The diameter information can also give insights into whether to switch the plan skeleton. For example, if our $B(K^d)$ is bad to the point that it does not contain the optimal value, then we should forgo it.

3 Monte-Carlo planning algorithm for continuous domains

3.1 Dealing with infeasible actions

if the next state node's state is equivalent to the current node's state, because the action failed, then if I simulate forward from this state, then I might get a different reward. If I get a high reward, then this might mean that current state is actually good, but it does not mean that that particular (s,a), which got you back to the same state s, was good. We would waste a lot of time updating the value of an edge that gets you back to the same state What if I make it so that it returns 0 if it the sampled action is infeasible? That is, treat (s,a) as a deadend-action. If I do this, then there is no recursion after this

3.2 An idea about the roll-out policy

It seems that the purpose of the roll-out policy is to supposed to give a rough first estimate of the untravelled edge (s, a). We might do a (fast) roll-out by postponing the calls to the motion planner when using this policy.

4 Improving Monte-Carlo planning with learning

Instead of initializing Q-values of new nodes with 0, we will learn Q off-line and then soft-update it online. Also, instead of using a uniform policy to sample actions, we will use a learned policy. If our learned policy does not have a support for the entire action space, then this will not guarantee probabilistic completeness. So, our approach would be to use a uniform policy with some small probability.

5 Theoretical conjectures

Suppose that our reward function is 1 at the goal state(s), and 0 otherwise. In this case, finding a satisficing solution would be equivalent to finding an optimal solution. A probabilistic completeness in this case would give us the guarantee that we will find this optimal solution, if we use a uniform random policy for π as the number of evaluations of edges reaches infinity. This, however, does not give us the convergence rate, which is much more desirable.

Most of the papers in the Monte-carlo planning literature discusses about how to partition the action space using a bandit algorithm, whether continuous or discrete, in order to show convergence rates. These algorithms, however, won't be able to scale to large dimensions. Hence our hope is to use learning, and show the convergence rate as a function of estimation error. Here are specific questions :

1. Suppose when I sample an action (line 10 of Algorithm 2), I sample from a uniform policy with probability p, and from a learned policy π_θ. If my learned policy has an error of δ, $|\pi(s) - \pi^*(s)| \leq \delta \ \forall s \in S$, then what is my convergence rate?

2. Suppose I have learned Q_α such that $|Q_\alpha(s,a) - Q^*(s,a)| \le \delta \; \forall (s,a) \in S \times A$. If in the backup step in line 18 of Algorithm 2, I use

$$Q(s,a) \leftarrow (1-q) \cdot Q_\alpha(s,a) + q \cdot Q_{MC}(s,a)$$

   where $Q_{MC}(s,a)$ is the value that I obtained by solely using Monte-Carlo rollouts, and $q \in [0,1]$, then what is my convergence rate?

6 Optimistic planning using the Voronoi region

We would like to find the optimal action x^* under the assumption that

$$f(x) - f(y) \le \lambda \cdot d(x,y)$$

So, we have $f(x) \le f(y) + \lambda \cdot d(x,y)$.

At any time point t, we know that we would have evaluated $x_1, \cdots x_t$ number of points. We denote the Vornoi regions induced by these points as $V(x_i)$. By the definition of the Voronoi region, we know that the lowest upper-bound of value of points in a region is provided by the furthest distance to the generator x_i of the region. Hence, we propose the following:

1. Compute the convex hulls of the Voronoi regions $V(x_1), \cdots, V(x_t)$. Denote $CH(V_i)$ as the convex hull of the i^{th} Voronoi region.

2. Select the most optimistic region by

$$i^* = \arg \max_{i \in 1,\cdots,t} f(x_i) + \max_{x \in CH(V_i)} d(x, x_i)$$

3. Randomly sample a point from $V(x_{i^*})$.

6.1 Computing the convex hull of the Voronoi region

Suppose we have points $p_1, \cdots, p_n$. Denote the line segment from p_i to p_j as $\overline{p_i p_j}$. Given two points p_i and p_j, by the definition of a Voronoi region, we know that the Voronoi regions whose sites are p_i and p_j are defined by the perpendicular bisector of the line segment $\overline{p_i p_j}$. We can find this analytically.

In general, if we have n points, then the Voronoi region of a point p_i can be determined by the intersection of the perpendicular bisectors of the line segment between p_i and all the other points. Finding the intersection of hyperplanes takes $O(n \log n)$ time, where n is the number of hyperplanes. If we repeat this for all the points, then it takes $O(n^2 \log n)$ time.

Once we have the intersection, we can the set of points where the hyperplanes that define the intersection meet. These give the convex hull.

6.2 Analysis

Denote $\delta(V_i) = \max_{x \in CH(V_i)} d(x, x_i)$. We know that if i^{th} Voronoi region contains x^*, then by our smoothness assumption it is bounded above by

$$f(x^*) \leq f(x_i) + \delta(V_i)$$

So, we will never evaluate a region whose upper bound is lower than $f(x^*)$, because

$$f(x_j) + \delta(V_j) \leq f(x^*) \leq f(x_i) + \delta(V_i)$$

We can adopt the similar assumptions from DOO yet more precise. We know at at any point time t, the diameters of the Voronoi region is decreasing. So, we have decreasing sequence of diameters, with respect to the number of evaluations t, $B(t)$.

7 Sampling-based approach

This scales poorly with the number of dimensions. New algorithm:

1. With probability ϵ, sample uniform randomly from $\mathcal{X}$
2. With probability $1 - \epsilon$, sample uniform randomly from the best Voronoi region $V(x_{i^*})$, $i^* = \arg\max_{i \in 1, \cdots, t} f(x_i)$

8 Geometric task-and-motion-planning problem

We denote the workspace of the robot as $\mathcal{W}$. A region $\mathcal{R}$ is defined as a subset of the workspace. We define *task constraint* as a set of objects to be packed in the ordered sequence of regions,

$$[\mathcal{R}_1 : (B_1^{(1)}, \cdots, B_{m_1}^{(1)}), \cdots, \mathcal{R}_T : (B_1^{(T)}, \cdots, B_{m_T}^{(T)})]$$

and the *connecting region* $\mathcal{R}_{connect}$ between two regions $\mathcal{R}_i$ and $\mathcal{R}_j$ as $\mathcal{R}_{connect} \not\subset \mathcal{R}_i \cup \mathcal{R}_j$ such that, from any robot configuration in $\mathcal{R}_{connect}$, any configuration in $\mathcal{R}_i$ and $\mathcal{R}_j$ can be reached, without considering the movable obstacles. We will call the regions specified in the task constraint as *key regions.*

The problem statement is as follows. *Given a task constraint, regions, and connecting regions between all pairs of regions, find the optimal motion plan to pack each object into its corresponding key region in the given order, where the quality of a plan is measured by the number of robot operations in it.* We have following variability in our problem:

- The objects specified within a key region may or may not be ordered.
- There may be other movable obstacles that are not specified in the task constraint.

We have following assumptions:

- We cannot find a feasible solution of the packing problem for $\mathcal{R}_t$ without finding one for $\mathcal{R}_{t-1}$.
- How we pack objects in $\mathcal{R}_{t-1}$ does not affect the feasibility of the packing problem for $\mathcal{R}_t$, but affects the optimality.

Given this setup, G-TAMP problem is a sequence of interrelated packing problems. A packing problem

$$\mathcal{R} : (B_1, \cdots, B_m)$$

consists of two parts: fetching objects from their initial regions, and then placing them in $\mathcal{R}$. We assume that we cannot move a target object once we pack it in $\mathcal{R}$, in order to remove redundant actions.

In fetching-and-placing a target object, we need to determine objects that are in the way and clear them out. Therefore, we can see that a packing problem has four unknowns, for each object:

1. Finding a robot reaching motion to the current target object from the robot's current configuration.
2. Finding a robot constraint-removal motion for clearing movable obstacles from this reaching-motion
3. Finding a robot placing motion that places the current target object in $\mathcal{R}$ such that all of target objects $B_1, \cdots, B_m$ can be packed in $\mathcal{R}$.
4. Finding a robot constraint-removal motion for clearing movable obstacles from this placing-motion

These problems are related in that we cannot solve for 2 without solving for 1, 3 without solving for 2, and so on. However, if we tackle this naively in this order, we may waste our effort finding solutions for 1, 2, and 3 for many early objects, only to find that these early placements prevents a solution for latter objects.

How should we define the sub-problem for which MCTS will be used for? We should define it such that a solution to the earlier sub-problem cannot have feasibility-level influence on the solution of the next sub-problem, and we need the solution to the sub-problem for the subsequent sub-problem. For example, we can separate object fetching problem and NAMO problem, because how I fetch objects cannot influence the feasibility of the NAMO problem, and I cannot solve (or even define) the NAMO problem without knowing the fetching path. But I cannot separate clearing each object within a NAMO problem because if I clear an earlier obstacle such that it makes the next obstacle unreachable, then the problem is infeasible.

I cannot define the reaching-problem for object B_2 if I have not solved the reaching-and-constraint-removal motion for object B_1.