CS 294-115 Algorithmic Human-Robot Interaction **Fall 2016**

Lecture 18: Recap

Scribes: Glen Chou, Nick Landolfi

18.1 Experimental Design

Treatments	Measures
Experimental Units	Assignment Method

Hopefully upon recalling our discussion of experimental design, you recall the phrase: "make factorial experiments!" We defined experiments as having the four components listed above. Treatments or conditions define your independent variables, those aspects which you change and control. Measures define you dependent variables, those aspects believed to be influenced by the independent variables. Experimental units are those things to which you assign treatments; often people but potentially motion planning problems or trajectory demonstrations in our context. Finally, the assignment method is the manner which you go about matching treatments to experimental units.

We discussed within-subject designs in which each subject receives all treatments, which helps control for variability across experimental units. However, in some cases (medicine) within subjects proves infeasible, and therefore a between subjects design prevails, in which each subject is assigned a single treatment.

Some key takeaways:

1. Extract your independent variables. It is harder in practice, but at least try to be aware of the entire factorial
2. Consider stating the full space of condition levels, and state why you can't perform the full factorial (logistical, feasibility, etc.)
3. When you see new algorithms being compared to a couple of baselines, try to identify the key components at play, and which conditions are perhaps untested.

Always state the motivation of your experiments, define a hypothesis, and, to the best of your ability, test as many combinations of conditions as possible.

18.2 Motion Planning

In the motion planning problem, we first define the configuration space C. In the configuration space, the robot becomes a point $q \in C$, in contrast to the potentially unwieldy set of points the robot inhabits the world space.

Assuming that we know the geometry of the robot in Figure 18.2, we know where the arm is in the world space given the parameters (θ_1, θ_2). Specifically, we can go from C-space to world space using the forward

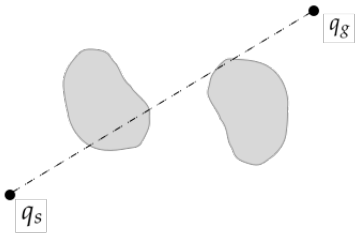

Figure 18.1: The motion planning problem

kinematics function ϕ_b. Given some point b on the robot, we can apply the forward kinematics mapping $\phi_b : q \to (x, y, z)$ to go from a configuration to some location in the world space. In general, obstacles may obstruct the simple straight line path, as can be seen in Figure 18.2.

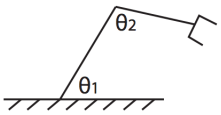

Figure 18.2: Robot arm with 2 degrees of freedom

To solve motion planning problems, we discussed a few methods:

18.2.1 Visibility Graphs

Visibility graphs, while optimal, operate under the assumptions that our obstacles are polygonal and that we have an explicit representation of the obstacles in C-space, C_{obs}. In general, however, we do not have access to such an explicit representation due to how expensive the computation is. Another downside is that the planned paths bring us very close to obstacles, so perhaps we may want to define "optimality" in a different sense, also incorporating a level of safety.

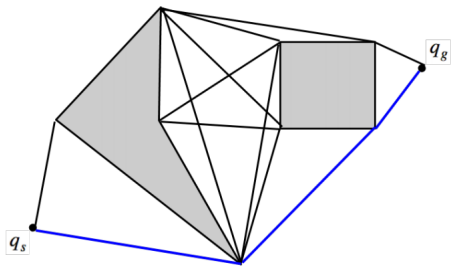

Figure 18.3: Example of a visibility graph

18.2.2 Grid Search

If we have access to a collision checker, which returns 0 if there is no collision and 1 otherwise, we can sidestep the constraints of polygonal obstacles and requirement of having an explicit representation of C_{obs} by using grid search. This technique discretizes C-space into a grid and performs graph search on the grid vertices, discarding them if they are in collision with obstacles. However, this method scales exponentially in the number of grid points.

18.2.3 Sampling

Sampling algorithms build a graph quickly, sampling the space until a point is found that can be added to the path without collisions. If no path can be found, we just keep sampling more until a path is found. Discussion on some specific sampling algorithms follows.

18.2.3.1 Probabilistic Roadmap (PRM)

In the PRM, we sample M collision-free milestones, including the start and goal states s, g if they are known. Two cases that the PRM can be used for are simple vs multiple query goals; in the former, there is just one query (q_s, q_g), and in the latter, the PRM deals with multiple queries $\{(q_s, q_g)^{(i)}\}_{i=1}^{N}$.

Next, the algorithm tries to connect the M milestone points. There are a few ways to connect these points; we can either try to connect all of the pairs of points, or we can connect each point to the k nearest points (k-PRM), or we can connect each point to points within some radius.

Then, we can do graph search (A^* or some other variant) to find a path. If we fail, we sample more milestones and rerun the algorithm.

18.2.3.2 [Bi-directional] Rapidly Exploring Random Tree (RRT)

We introduced RRTs as a special case of PRMs (although the initial paper did not introduce them as such). In RRTs you maintain a single connected component, and sample one milestone at a time, only connecting it to the closest visible point in your connected component of previously sampled points. Visible points are those which can be connected by a line. Therefore we are actually just building a tree of points from out samples, at each stage checking whether we can also connect to the goal state.

The bi-directional variant maintains two connected components, one for those points connected to the start configuration and one for those connected to the goal configuration. For each sample, we try to connect it both to the start and goal connected components. Again we use closest "visible" (non-obstructed linear path) neighbor for parent selection. The algorithm terminates when a sampled point can be connected to both the start and goal connected components.

In both instances, you recover the path by performing graph search on the accumulated connected component, which should be faster than searching on a general graph, because we maintained the invariant that our component is a tree. In practice running generic RRTs results in potentially long, odd and jerky motion paths in world space. Therefore post-processing methods, such as shortcutting are performed on the returned path in an attempt to shorten and/or smooth it.

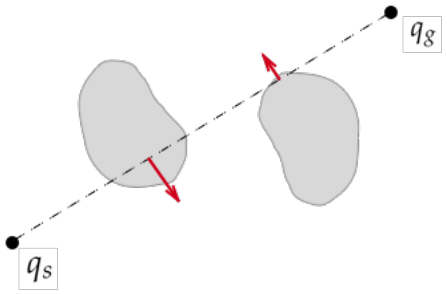

Figure 18.4: The Euclidean gradient $\nabla_\zeta U$ pushes the trajectory away from the obstacles

18.2.3.3 RRT*

RRT* is an asymptotically optimal variant of RRTs which introduces two key changes to the traditional algorithm:

1. **Parent selection:** RRT* selects the parent in the connected components based on the total backward cost to the sampled point. Rather than greedily based on the closest point in the connected component.
2. **Re-wiring:** after a new sample, RRT* rewires the connected component by setting the sample point to be the parent of any nodes whose backward cost would decrease by making this change.

In practice, traditional bi-directional RRTs are the most popular variant, and despite its theoretical interest, RRT* sees little use.

18.3 Trajectory Optimization

We define a trajectory ζ mapping time to C space: $\zeta : [0, T] \to C$. Our definition of a trajectory defines where we want the robot to be at a given time.

We also define the cost functional $\mathcal{U} : \Xi \to \mathbb{R}^+$, where the cost can integrate many factors such as path length, smoothness, and legibility. For example, we might want to use smoothness in order to better match what people expect the robot to do.

However, using the standard Euclidean norm as a distance metric when optimizing for smoothness is not that great of an idea, as it will consider a trajectory with a large perturbation at one point to be preferable to a trajectory with that same perturbation but smoothed out over the remainder of the path. In particular, we can change our metric to prefer smoothness by defining the inner product between two trajectories $< \zeta_1, \zeta_2 >= \zeta_1^T \zeta_2$ to be $< \zeta_1, \zeta_2 >= \zeta_1^T A \zeta_2$, where A is a tridiagonal positive semidefinite matrix.

In this case, we will perform gradient descent using Equation 18.1:

$$\zeta_{i+1} = \zeta_i - \frac{1}{\alpha} A^{-1} \nabla_\zeta \mathcal{U} \tag{18.1}$$

One particular obstacle cost function that we can use is:

$$\mathcal{U}_{obs}[\zeta] = \int_t \int_b c(\phi_b(\zeta(t)))dbdt \tag{18.2}$$

Here, b is the set of body points on the robot, t is time, $c(\cdot)$ is a cost function that quantifies the distance from the point to the closest obstacle (in the case of CHOMP, a signed distance field is used for $c(\cdot)$), and $\phi_b(\cdot)$ is the forwards kinematics map.

18.4 Learning from Demonstration

In learning from demonstration, we want to find an optimal cost function. This is motivated by our desire to learn human preferences, expectations, and learning skills.

We define inverse reinforcement learning / inverse optimal control. In this framework, we want to find a mapping $\zeta_D \to \mathcal{U}$, where ζ_D is some demonstrated trajectory. In IOC, we want to find the cost function $\mathcal{U}$ that best explains the demonstration.

18.4.0.1 Maximum Margin Planning

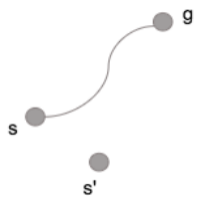

Figure 18.5: Finding a new trajectory

Given a demonstrated trajectory ζ_D from a start state s to the goal state g, we may want to instead find the trajectory $\hat{\zeta}$ going from a new start state $\hat{s}$ to g (Figure 18.5), where we define

$$\hat{\zeta} = \arg\min_{\zeta} \mathcal{U}(\zeta) \tag{18.3}$$

We want $\mathcal{U}(\zeta_D) \leq \min_{\zeta} \mathcal{U}(\zeta)$. This problem has a trivial solution $\mathcal{U}(\zeta) = k$. To bypass this, we can modify the IOC problem to read as follows

$$\mathcal{U}(\zeta_D) \leq \min_{\zeta}(\mathcal{U}(\zeta) - \ell(\zeta, \zeta_D)) \tag{18.4}$$

We want 18.4 to be satisfied with the largest margin possible, so we can formulate the maximum-margin planning problem as

$$\min_{\mathcal{U}}(\mathcal{U}(\zeta_D) - (\min_{\zeta}(\mathcal{U}(\zeta) - \ell(\zeta, \zeta_D)))) \tag{18.5}$$

Instead of searching over all possible cost functionals $\mathcal{U}$, we can choose $\mathcal{U}$ to be linear in the features: $\mathcal{U}(zeta) = w^\top f(\zeta)$. This results in a gradient descent equation of

$$w_{i+1} = w_i - \frac{1}{\alpha}(f(\zeta_D) - f(\zeta^*(w_i)) \tag{18.6}$$

where $f(\zeta^*(w_i))$ is the current optimum.

18.4.0.2 Maximum Entropy IRL

In maximum entropy inverse reinforcement learning we do not assume that our demonstrations are optimal, but instead assume that the demonstrated trajectories are sampled (noisily) from some distribution of trajectories. Therefore we seek a model of the probability of a trajectory given some true parametrization of our reward function (which we still assume is linear in the features). Explicitly, we choose the maximum entropy (exponential) family distribution, which encodes the least information while respecting the optimality of the demonstrated trajectories in expectation:

$$P(\zeta_D \mid w) \propto \exp\left(-w^T f(\zeta_D)\right) \tag{18.7}$$

With this model in hand (called a Boltzmann distribution), we perform maximum likelihood estimation to obtain w. Observe:

$$\max_w \quad P(\zeta_D \mid w)$$

is equivalent to maximizing the log likelihood, given by:

$$\log P(\zeta_D \mid w) = \log \frac{\exp\left(-w^T f(\zeta_D)\right)}{\int_\zeta \exp\left(-w^T f(\zeta)\right)} = -w^T f(\zeta_D) - \log \int_\zeta \exp\left(-w^T f(\zeta)\right)$$

We take the gradient with respect to w:

$$\nabla_w = -f(\zeta_D) - \frac{1}{\int_\zeta \exp\left(-w^T f(\zeta)\right)} \int_\zeta -f(\zeta) \exp\left(-w^T f(\zeta)\right)$$

A key insight presents itself if you recognize the second term as an expectation:

$$\nabla_w = -f(\zeta_D) + E\left[f(\zeta)\right] \tag{18.8}$$

We perform gradient *ascent*:

$$w_{i+1} = w_i + \frac{1}{\alpha}\left(E\left[f(\zeta)\right] - f(\zeta_D)\right) \tag{18.9}$$