Predicting Partial Paths from Planning Problem Parameters

Sarah Finney, Leslie Kaelbling, Tomás Lozano-Pérez
CSAIL, MIT
email: sjf,lpk,tlp@csail.mit.edu

***Abstract*— Many robot motion planning problems can be described as a combination of motion through relatively sparsely filled regions of configuration space and motion through tighter passages. Sample-based planners perform very effectively everywhere but in the tight passages. In this paper, we provide a method for parametrically describing workspace arrangements that are difficult for planners, and then learning a function that proposes partial paths through them as a function of the parameters. These suggested partial paths are then used to significantly speed up planning for new problems.**

I. Introduction

Modern sample-based single-query robot-motion planners are highly effective in a wide variety of planning tasks. When they do encounter difficulty, it is usually because the path must go through a part of the configuration space that is constricted, or otherwise hard to sample.

One set of strategies for solving this problem is to approach it generically, and to develop general methods for seeking and sampling in constricted parts of the space. Because these planning problems are ultimately intractable in the worst case, the completely general strategy cannot be made efficient. However, it may be that a rich set of classes of more specific cases can be solved efficiently by learning from experience.

The basic question addressed in this paper is whether it is possible to learn a function that maps directly from parameters describing an arrangement of obstacles in the workspace, to some information about the configuration space that can be used effectively by a sample-based planner. We answer this question in the affirmative and describe an approach based on task templates, which are parametric descriptions (in the workspace) of the robot's planning problem. The goal is to learn a function that, given a new planning problem, described as an instance of the template, generates a small set of suggested partial paths in the constricted part of the configuration space. These suggested partial paths can then be used to "seed" a sample-based planner, giving it guidance about paths that are likely to be successful.

Task templates can be used to describe broad classes of situations. Examples include: moving (all or part of) the robot through an opening and grasping something; moving through a door carrying a large object; arriving at a pre-grasp configuration with fingers "straddling" an object. Note that the task template parameters are in the workspace rather than the configuration space of the robot, so knowing the parameters is still a long way from knowing how to solve the task.

One scenario for obtaining and planning with task templates would be a kind of teleoperation via task-specification. A human user, rather than attempting to control the robot's end-effector or joints directly, would instead choose an appropriate task template, and "fit" it to the environment by, for instance, using a graphical user interface to specify the location of a car-window to be reached through and a package on the car seat to be grasped. Given this task-template instance the planner can generate a solution and begin to execute it. During the process of execution the operator might get additional sensory information (from a camera on the arm, for example), and modify the template dynamically, requiring a very quick replanning cycle. Eventually, it might be possible to train a recognition system, based on 2D images or 3D range data to select and instantiate templates automatically.

Of course, we cannot get something for nothing: the power of learning comes from exploiting the underlying similarity in task instances. For this reason, template-based learning cannot be a general-purpose solution to motion planning. It is intended to apply to commonly occurring situations where a very fast solution is required. Although one might be tempted to encode a general workspace as a single task template, with a finite (but very large) set of parameters that indicate which voxels are occupied, there is no reason to believe that the resulting learning problem for such a template is tractable.

II. Related work

A good deal of work has been done on improving multiple-query roadmaps by biasing their sampling strategy to generate samples in difficult areas of configuration space and avoid over-sampling in regions that can be fairly sparsely covered. Several techniques involve rejection sampling intended to concentrate the samples in particular areas, such as near obstacles [14], or near configuration-space bottlenecks [4]. Pushing samples toward the medial axis of the configuration freespace is another method for generating samples that are concentrated in the difficult narrow passages [7]. Visibility-based approaches also strive to increase sampling near bottlenecks [13].

Each of these techniques involves computationally expensive operations, as they are intended to be used in a multiple-query setting, in which the cost is amortized over a large number of queries within a static environment. We are focused instead on a single-query setting, in which these techniques are prohibitively time-consuming. However, the collective body

of work argues persuasively that a relatively small number of carefully selected samples can make the planning job much easier. Our approach is similarly intended to address these difficult regions of the planning space.

Other similarly motivated techniques temporarily expand the freespace by contracting the robot to identify the narrow passages [11, 6], and these methods are fast enough to apply to single-query problems. Nonetheless, they are required to discover the structure of the environment at planning time. We would like to use learning to move much of this work offline.

A related class of techniques looks at aspects of the workspace, rather than configuration space, to bias sampling toward finding difficult narrow passages [8, 16, 15, 5]. This work builds on the intuition that constrained areas in configuration space are also often constrained in the workspace, and are easier to identify in the lower dimensional space. Our work shares this intuition.

There are also a number of strategies for guiding the search in a single-query setting. Voronoi-based [17] and entropy-guided [2, 10] exploration are among these techniques. Since we are proposing a learning method that predicts partial paths in order to help a single-query planner, our approach is compatible with any single-query probabilistic roadmap exploration strategy. In particular, in this paper we integrate our approach with Stanford's single-query, bidirectional, lazy (SBL) planner [12].

Other approaches have also taken advantage of learning. Features of configuration space can be used to classify parts of the space and thereby choose the sampling strategies that are most appropriate for building a good multiple-query roadmap in that region [9]. Burns and Brock learn a predictive model of the freespace to better allocate a roadmap's resources and avoid collision checking [1]. This use of learning is related to ours, but is again in a multiple-query setting, and so does not address generalizing to different environments.

III. Task-template planning

The task-template planner is made up of two phases: first, the task is described parametrically as an instance of the template, and used as input to a learned *partial-path suggestion function* that generates a set of candidate partial paths for solutions; second, these partial paths are used to initialize a sample-based planner, suitably modified to take advantage of these suggestions. If the suggested partial paths are indeed in parts of the configuration space that are both useful and constricted, then they will dramatically speed up the sample-based planner.

To make the discussion concrete, we will use as a running example a mobile two-degree-of-freedom arm, operating in essentially a planar environment. One common task for such a robot will be to go through doors. A door in a planar wall can be described using 3 parameters (x_{door}, y_{door}, θ_{door}) as shown in figure 1 (top left). Note that the environment need not be exactly conformant to the template, as shown in figure 1 (top right). As long as the suggestions to the planner help it with the difficult parts of the environment, the planner can handle other sparsely arranged obstacles. The remaining panels in figure 1 show some useful suggested partial paths.

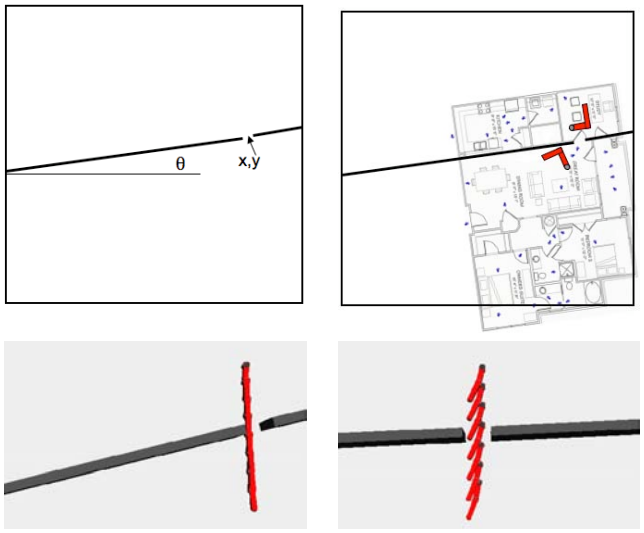

Fig. 1. A task template. Top left: Parameterization of the template. Top right: Application of task template to a complex environment. Bottom: Suggested partial paths generated by the same strategy for different environments.

The overall learning approach generates training examples by calling a single-query planner on a number of planning problems, each of which is an instance of the same task template. Each plan is then examined to extract the most constrained portion of the path, and the partial path through this "tight spot" is stored along with the task-template parameters describing the world. We then train several partial-path generators on this data. The idea is that the partial-path suggestion function will learn the parametric dependence of the useful partial paths on the task-template instance parameters, and therefore apply to previously unseen examples.

Additionally, we must adapt the single-query planner so that it can take in, along with the start and goal configuration for a particular query, a set of suggested partial paths.

A. Generating training data

For each task template, we assume a source of training task instances of that template. These instances might be synthetically generated at random from some plausible distribution, or be drawn from some source of problems encountered in the world in which the robot will be operating.

We begin with a set of task instances, $t^1, \ldots, t^n$, where each task instance is a specification of the detailed problem, including start and goal configurations and a description of the workspace obstacles. These descriptions will generally be non-parametric in the sense of not having a fixed size. We will convert each of these tasks and their solution paths into a training example, $\langle x^i, y^i \rangle$, where x^i is a parametric representation of t^i as an instance of the task template, and y^i is a path segment. Intuitively, we instantiate the task template for the particular task at hand. In the case of the

doorway example this means identifying the door, and finding its position and orientation, x_{door}, y_{door} and θ_{door}.

Currently, we assume the each t^i in the training set is "labeled" with a parametric description x^i; in the future, we expect that the labels might be computed automatically. To generate the y^i values, we begin by calling the single-query planner on task instance t^i. If the planner succeeds, it returns a path $p = \langle c_1, ..., c_r \rangle$ where the c's are configurations of the robot, connected by straight-line paths.

We do not, however, want to use the entire path to train our learning algorithm. Instead we want to focus in on the parts of the path that were most difficult for the planner to find, so we will extract the most constrained portion of the path, and use just this segment as our training data. Before we analyze the path, we would like to reduce the number of unnecessary configurations generated by the probabilistic planner, so we begin by smoothing the path. This process involves looking for non-colliding straight-line replacements for randomly selected parts of the path, and using them in place of the original segment. We made use of the SBL planner's built-in smoother to do this.

We also want to sample the paths at uniform distance intervals, so that paths that follow similar trajectories through the workspace are similar when represented as sample sequences. Thus, we first resample the smoothed path at a higher resolution, generating samples by interpolating along p at some fixed distance between samples, resulting in a more finely sampled path.

Given this new, finely sampled path p', we now want to extract the segment that was most constrained, and therefore difficult for the planner to find. In fact, if there is more than one such segment, we would like to find them all. For each configuration in the path, we draw some number of samples from a normal distribution centered on that configuration (sampling each degree of freedom independently). Each sample is checked for collision, and the ratio of colliding to total samples is used as a measure of how constrained the configuration is.

We then set a threshold to determine which configurations are tight, and find the segments of contiguous tight configurations. Our outputs need to have a fixed length l, so for all segments less than l, we pad the end with the final configuration. For segments with length greater than l, we skip configurations in cases where skipping does not cause a collision.

In our more complicated test domains, we found that it was useful to add a step that pushes each configuration away from collisions, by sampling randomly around each configuration in the path, and replacing it if another is found that has greater clearance. This makes it less likely that we will identify spurious tight spots, and it also makes the segment we extract more likely to generalize, since it has a larger margin of error. This step also requires that we are careful not to decrease the path's clearance when we smooth it. In this way, we extract each of the constrained path segments from the solution path. Each of these, paired with the parametric description of the task, becomes a separate data point.

B. Learning from successful plans

Each of our n training instances $\langle x^i, y^i \rangle$ consists of a task-template description of the world, x^i, described in terms of m parameters, and the constrained segment from a successful path, y^i, as described above. The configurations have consistent length d (the number of degrees of freedom that the robot has), and each constrained segment has been constructed to have consistent length l, therefore y^i is a vector of length ld.

At first glance, this seems like a relatively straightforward non-linear regression problem, learning some function f^* from an m-dimensional vector x to an ld-dimensional vector y, so that the average distance between the actual output and the predicted output is minimized. However, although it would be useful to learn a single-valued function that generates one predicted y vector given an input x, our source of data does not necessarily have that form. In general, for any given task instance, there may be a large set of valid plans, some of which are qualitatively different from one another. For instance, members of different homotopy classes should never be averaged together.

Consider a mobile robot going around an obstacle. It could choose to go around either to the left of the obstacle, or to the right. In calling a planner to solve such problems, there is no general way to bias it toward one solution or the other; and in some problems there may be many such solutions. If we simply took the partial paths from all of these plans and tried to solve for a single regression function f from x to y, it would have the effect of "averaging" the outputs, and potentially end up suggesting partial paths that go through the obstacle.

1) Mixture model: To handle this problem, we have to construct a more sophisticated regression model, in which we assume that data are actually drawn from a mixture of regression functions, representing qualitatively different "strategies" for negotiating the environment. We learn a generative model for the selection of strategies and for the regression function given the strategy, in the form of a probabilistic mixture of h regression models, so that the conditional distribution of an output y given an input x is

$$\Pr(y|x) = \sum_{k=1}^{h} \Pr(y|x, s = k) \Pr(s = k|x) \quad .$$

where s is the mixture component responsible for this example.

For each strategy, we assume that the components of the output vector are generated independently, conditioned on x; that is, that

$$\Pr(y|x, s = k) = \prod_{j=1}^{ld} \Pr(y_j|x, s = k) \quad .$$

Note that this applies to each configuration on the path as well as each coordinate of each configuration. Thus, the whole partial path is being treated as a point in an ld-dimensional space and not a sequence of points in the configuration space. The parameter-estimation model will cluster paths such

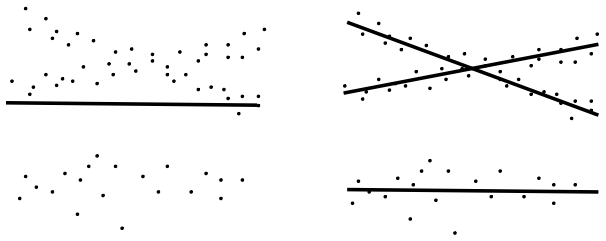

Fig. 2. Data with linear regression (left) and mixture regression (right) fits.

that this independence assumption is satisfied, to the degree possible, in the model.

Then, we assume that each y_j has a linear dependence on x with Gaussian noise, so that

$$\Pr(y_j|x, s=k) = \frac{1}{\sigma_{jk}\sqrt{2\pi}} \exp\left(-\frac{(y_j - w_{jk}\cdot x)^2}{2\sigma_{jk}^2}\right) \ ,$$

where σ_{jk} is the standard deviation of the data in output dimension j of strategy k from the nominal line, and w_{jk} is an m-dimensional vector of weights specifying the dependence of the mean of the distribution of y_j in strategy k given x as the dot product of w_{jk} and x.

It would be possible to extend this model to contain non-linear regression models for each mixture components, and it might be useful to do so in the future. However, the current model can approximate a non-linear strategy function by using multiple linear components.

In our current model, we assume that $\Pr(s = k|x)$ is actually independent of x (we may wish to relax this in future), and define $\pi_k = \Pr(s = k)$. So, finally, we can write the log likelihood of the entire training set $LL(\sigma, w, \pi)$, as a function of the parameters σ, w, and π, (note that each of these is a vector or matrix of values), as

$$\sum_{i=1}^{n} \log \sum_{k=1}^{h} \pi_k \prod_{j=1}^{ld} \frac{1}{\sigma_{jk}\sqrt{2\pi}} \exp\left(-\frac{(y_j^i - w_{jk}\cdot x^i)^2}{2\sigma_{jk}^2}\right) \ .$$

We will take a simple maximum likelihood approach and attempt to find values of σ, w, and π that maximize this likelihood, and use that parameterization of the model to predict outputs for previously unseen values of x.

Figure 2 illustrates a data set in which the x and y are one dimensional. In the first frame, we show the best single linear regression line, and in the second frame, a mixture of linear regression models. It is clear that a single linear (or non-linear, for that matter) regression model is inappropriate for this data.

The model described in this section is essentially identical to one used for clustering trajectories in video streams [3], though their objective is primarily clustering, where ours is primarily regression.

2) Parameter estimation: If we knew which training points to assign to which mixture component, then the maximum likelihood parameter estimation problem would be a simple matter of counting to estimate the π_k and linear regression to estimate w and σ. Because we don't know those assignments, will will have to treat them as hidden variables. Let $\gamma_k^i = \Pr(s^i = k|x^i, y^i)$ be the probability that training example i belongs to mixture component k. With this model we can use the expectation-maximization (EM) algorithm to estimate the maximum likelihood parameters.

We start with an initial random assignment of training examples to mixture components, ensuring that each component has enough points to make the linear regression required in the M step described below be well-conditioned.

In the expectation (E) step, we temporarily assume our current model parameters are correct, and use them and the data to compute the responsibilities:

$$\gamma_k^i := \frac{\pi_k \Pr(y^i|x^i, s^i = j)}{\sum_{a=1}^{h} \pi_a \Pr(y^i|x^i, s^i = a)} \ .$$

In the maximization (M) step, we temporarily assume the responsibilities are correct, and use them and the data to compute a new estimate of the model parameters. We do this by solving, for each component k and output dimension j, a weighted linear regression, with the responsibilities γ_k^i as weights. Weighted regression finds the weight vector, w_{jk}, minimizing

$$\sum_i \gamma_k^i (w_{jk}\cdot x^i - y_j^i)^2 \ .$$

When the regression is numerically ill-conditioned, we use a ridge parameter to regularize it.

In addition, for each mixture component k, we re-estimate the standard deviation:

$$\sigma_{jk} := \frac{1}{\sum_i \gamma_j^i} \sum_i \gamma_j^i (w_{jk}\cdot x^i - y_j^i)^2 \ .$$

Finally, we reestimate the mixture probabilities:

$$\pi_j := \frac{1}{n} \sum_{i=1}^{n} \gamma_j^i \ .$$

This algorithm is guaranteed to find models for which the log likelihood of the data is monotonically increasing, but has the potential to be trapped in local optima. To ameliorate this effect, our implementation does random re-starts of EM and selects the solutions with the best log likelihood.

The problem of selecting an appropriate number of components can be difficult. One standard strategy is to try different values and select one based on held-out data or cross-validation. In this work, since we are ultimately interested in regression outputs and not the underlying cluster structure, we simply use more clusters than are likely to be necessary and ignore any that ultimately have little data assigned to them.

C. Generating and using suggestions

Given the model parameters estimated from the training data, we can generate suggested partial paths from each strategy, and use those to bias the search of a sample-based planner. In our experiments, we have modified the SBL planner to accept these suggestions, but we expect that most sample-based planners could be similarly modified. We describe the way in which we modified the SBL planner below.

For a new planning problem with task-template description x, each strategy k generates a vector $y^k = w_k \cdot x$ that represents a path segment. However, in the new planning environment described by x, the path segment may have collisions. We want to avoid distracting the planner with bad suggestions, so we collision-check each configuration and each path between consecutive configurations, and split the path into valid subpaths. For example, if y is a suggested path, consisting of configurations $\langle c_1, \ldots, c_{15} \rangle$, imagine that configuration c_5 collides, as do the paths between c_9 and c_{10} and between c_{10} and c_{11}. We would remove the offending configurations, and split the path into three non-colliding segments: $\langle c_1 \ldots c_4 \rangle$, $\langle c_6 \ldots c_9 \rangle$, and $\langle c_{11} \ldots c_{15} \rangle$. All of the non-colliding path suggestions from each strategy are then given to the modified SBL planner, along with the initial start and goal query configurations.

The original SBL planner searches for a valid plan between two configurations by building two trees: T_s, rooted at the start configuration, and T_g, rooted at the goal configuration. Each node in the trees corresponds to a robot configuration, and the edges to a linear path between them. At each iteration a node n, from one of the trees is chosen to be expanded.[1] A node is expanded by sampling a new configuration n_{new} that is near n (in configuration space) and collision-free. In the function CONNECTTREES, the planner tries to connect T_s and T_g via n_{new}. To do this, it finds the node n_{close} in the other tree that is closest to n_{new}. If n_{close} and n_{new} are sufficiently close to one another, a candidate path from the start to the goal through the edge between n_{new} and n_{close} is proposed. At this point the edges along the path are checked for collision. If they are all collision-free, the path is returned. If an invalid edge is found, the path connecting the two trees is broken at the colliding edge, possibly moving some nodes from one tree to the other.

We extended this algorithm slightly, so that the query may include not only start and goal configurations, c_s and c_g, but a set of h path segments, $\langle p_1, ..., p_h \rangle$, where each path segment p_i is a list of configurations $\langle c_{i1}, ..., c_{ir} \rangle$. Note that r may be less than l, since nodes in collision with obstacles were removed. We now root trees at the first configuration c_{i1} in each of the path segments p_i, adding h trees to the planner's collection of trees. The rest of the configurations in each path segment are added as linear descendants, so each suggestion tree starts as a trunk with no branches.

Importantly, the order of the suggestions (barring those that were thrown out due to collisions) is preserved. This means that the suggested path segments are potentially more useful than a simple collection of suggested collision-free configurations, since we have reason to believe that the edges between them are also collision-free.

The original SBL algorithm chose with equal probability to expand either the start or goal tree. If we knew that the suggestions were perfect, we would simply require them to be connected to the start and goal locations by the planner. However, it is important to be robust to the case in which some or all of the suggestions are unhelpful. So, our planner chooses between the three *types* of trees uniformly: we choose the start tree and goal tree each with probability $1/3$, and one of the k previous path trees with probability $1/(3k)$. When we have generated a new node, we must now consider which, if any, of our trees to connect together. Algorithms 1 and 2 show pseudo-code for this process. The overall goal is to consider, for each newly sampled node n_{new}, any tree-to-tree connection that n_{new} might allow us to make, but still to leave all collision checking until the very end, as in the SBL planner.

T_{expand} is the tree that was just expanded with the addition of the newly sampled node n_{new}. If T_{expand} is either the start or goal tree (T_s and T_g respectively), we first try to make a connection to the other endpoint tree, through the CONNECTTREES function in the SBL planner. If this function succeeds, we have found a candidate path and determined that it is actually collision free, so we have solved the query and need only return the path.

If we have not successfully connected the start and goal trees, we consider making connections between the newly expanded tree and each of the previous path trees, T_i. The MERGETREES function tries to make these connections. The paths between these trees are not checked for collisions at this point. As mentioned above, the original algorithm has a distance requirement on two nodes in different trees before the nodes can be considered a link in a candidate path. We use the same criterion here. If the new node is found to be close enough to a node in another tree, we reroot T_i at node n_{close}, and then graft the newly structured tree onto T_{expand} at node n_{new}. If one connection is successfully made, we consider no more connections until we have sampled a new node.

Algorithm 1 TRYCONNECTIONS(n_{new}, T_{expand}) : Boolean

```
1:  success ← false
2:  if T_expand == T_g then
3:     success ← CONNECTTREES(n_new, T_expand, T_s)
4:  else if T_expand == T_s then
5:     success ← CONNECTTREES(n_new, T_expand, T_g)
6:  else
7:     for previous path trees T_i ≠ T_expand do
8:        merged ← MERGETREES(n_new, T_expand, T_i)
9:        if merged then
10:          break
11:       end if
12:    end for
13: end if
14: return success
```

IV. EXPERIMENTS

We have experimented with task-templates in several different types of environment, to show that learning generalized paths in this way can work, and that the proposed partial paths

[1]The SBL planning algorithm has many sophisticated details that make it a high-performance planner, but which we will not describe here, such as how to decide which configuration to expand. While these details are crucial for the effectiveness of the planner, we did not alter them, and so omit them for the sake of brevity.

Algorithm 2 MERGETREES($n_{new}, T_{expand}, T_{elim}$) : Boolean

1: $n_{close} \leftarrow T_{elim}$.CLOSESTNODE($n_{new}$)
2: $success \leftarrow$ CLOSEENOUGH(n_{close}, n_{new})
3: **if** $success$ **then**
4: REROOTTREE(n_{close}, T_{elim})
5: GRAFTTREE($n_{new}, n_{close}, T_{expand}, T_{elim}$)
6: **end if**

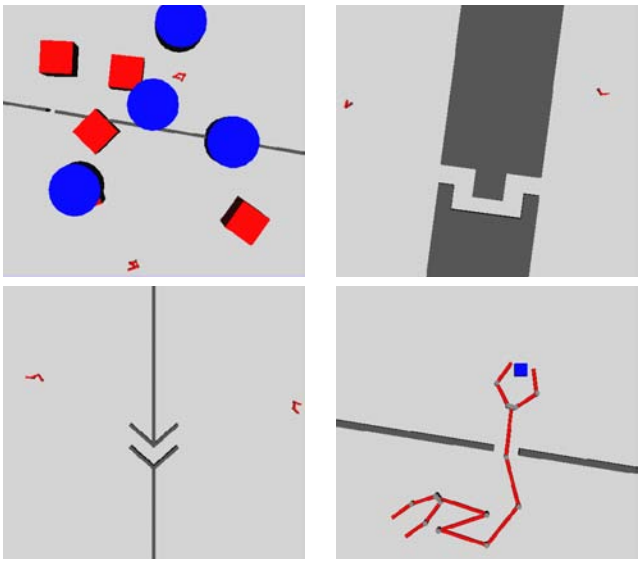

Fig. 3. Four types of environment for experimental testing. The robot shown in the first two illustrations is a planar mobile arm with varying number of degrees of freedom; in the last, it is a 8-DOF planar arm, with a simple hinged gripper.

can be used to speed up single-query planning. The simplest environment is depicted in figure 1, and the rest are shown in figure 3. Each illustration shows the obstacle placement and the start and goal locations for a particular task instance.

The first environment (depicted in figure 1, bottom) is simply a wall with a doorway in it. The robot is a mobile arm with a variable number of links in the arm, so that we can carefully explore how well the algorithms scale. Experiments done in this very simple domain were intended to be a proof of concept for both the generalized path learning from probabilistically generated paths, and the integration of suggested paths with the SBL planner.

The second domain extends the doorway domain slightly by introducing random obstacles during testing. The task template for this domain is the same as for the previous one, and in both cases training data was gathered in the absence of obstacles. Since randomly placing obstacles may completely obscure the door, we report planning times only for problems for which either the planner alone, or the modified planner, was able to find a solution.

The third domain was designed to show that the learning algorithm can learn more complicated path segments. This domain is tricky because the constrained portion is longer than in the previous examples. This means the learning is more difficult, as the training data is less well-aligned than in the previous cases, where the "tight spot" is generally easy to identify.

domain	average planner time	average time with suggestions
4 DOF door	1.5 s	0.7 s
5 DOF door	6.6 s	0.7 s
6 DOF door	38.1 s	1.0 s
7 DOF door	189.9 s	1.6 s
6 DOF cluttered door	82.3 s	5.3 s
4 DOF zig-zag corr.	12.9 s	2.3 s
5 DOF angle corr.	25.8 s	1.7 s
8 DOF simple gripper	78.6	52.0 s
8 DOF simple gripper with 2000 data points	74.8 s	13.7 s

Fig. 4. Planning performance results.

The first three domains all share the attribute that there exists a global rigid transform which could be applied to a good path segment for one task instance to generate a good path segment for a new task instance. This is not true of motion planning tasks in general, and our approach does not rely on the existence of such a transform; nonetheless, the fourth domain was designed to show that the technique applies in cases for which such a transform does not exist.

The final domain involves a robot with different kinematics, that has to have a substantially different configuration in going through the tight spot from that at the goal.

A summary of the results in each domain is shown in figure 4. We compare the time spent by the basic SBL planner to the time required by the partial-path-suggestion method, applying each to the same planning problem. Each planning time measurement was an average of 10 training trials, in which the model was trained on 500 data points (except the last experiment, which used 2000 data points), and then tested, along with the unmodified planner, on 100 planning problems. The running times reported throughout the paper for the version of the planner with suggestions includes the time required to generate suggestions for the new problem instance.

Figure 5 shows the time required to extract the constrained portion of each path in order to generate training data, and the time required to train all of the randomly initialized EM models. We used 500 samples with a sample variance of 0.01 for estimating the tightness of each configuration. The zig-zag corridor, angle corridor and simple gripper domains use the additional step for increasing collision clearance during the tight spot identification, which makes the process substantially slower. We used 20 random restarts of the EM algorithm. We discuss the details of each domain below.

A. The door task

The doorway in this domain, and the following one, is parameterized by the x, y position of the door, and the angle of

domain	time to find tight spots	time to train models
4 DOF door	1086 s	204 s
5 DOF door	1410 s	221 s
6 DOF door	2043 s	283 s
7 DOF door	3796 s	321 s
4 DOF zig-zag corr.	14588 s	1003 s
5 DOF angle corr.	25439 s	928 s
8 DOF simple gripper	42142 s	3396 s

Fig. 5. Offline learning time.

the wall relative to horizontal. We chose a tight spot segment length of 7 for this domain.

These simple experiments show that the suggestions can dramatically decrease the running time of the planner, with speed-ups of approximately 2, 9, 38 and 118 for the different number of degrees of freedom. For the robots with 4, 5, and 6 degrees of freedom, both methods returned an answer for every query. In the 7-DOF case, the unmodified planner found a solution to 88% of the queries, while the suggestions allowed it to answer 100% successfully within the allotted number (100,000) of iterations.

B. The door task with obstacles

We also did limited experiments by adding obstacles to the test environments after training on environments with just the door described above. The goal of these experiments was to show that the suggestions generated by the learner are often helpful even in the presence of additional obstacles.

For a mobile arm robot with 6 degrees of freedom, the suggestions allow the planner to find a solution approximately 15 times faster than the planner alone, on problems where either method finds an answer. The unmodified planner fails to find a solution over 4% of the time, whereas the suggestion-enhanced planner fails less just 0.3% of the time.

C. The zig-zag corridor

The zig-zag corridor domain was designed to require more interesting paths in order to test the robustness of the parameterized path learning. The robot for this domain has 4 degrees of freedom. The domain has two parameters, the vertical displacement of the corridor, and its rotation from vertical. The rotation parameter is encoded as three separate entries in the input vector: the angle, its sine and its cosine, since each of these may be necessary for the path to be expressed as a linear function of the input. The shape of the corridor is constant. The tight spot segment length is 15.

In this domain, we found that our results were initially unimpressive, due to difficulty in identifying the tight spot in our training paths. The step of pushing paths away from collisions before extracting the tight spot allowed us to improve our results to the same level as those achieved by hand-picking the tight spot. With the improved data generated in this way, we find that the speedups are substantial, but less dramatic than the doorway cases. Figure 6 shows an example

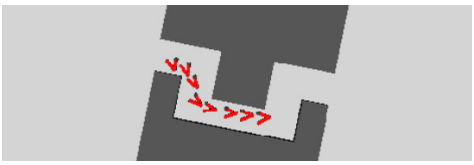

Fig. 6. A suggested path segment for the zig zag corridor.

suggestion for this domain. It is clear that the suggestions are providing useful guidance to the planner

If the corridor is no longer allowed to rotate, leading to a one-dimensional input parameter vector, we find that planning time with suggestions is reduced to 1.2 seconds. This suggests that we have increased the difficulty of the planning problem by adding the additional parameters.

D. The angle corridor

The angle corridor domain was designed to be a simple domain that has the property that there is not a single rigid transformation for transforming paths for one task instance into paths for another task instance. In this domain, the robot has 5 degrees of freedom. The task parameters are the vertical displacement of the corridor, and the angle of the bend in the middle of the corridor, which varied from horizontal (0°) by up to $\pm 50^\circ$. We again see a speedup of roughly 15 times. This demonstrates that our technique does not require that a rigid path transformation exist in order to make useful suggestions in new environments based on experience.

E. Simple gripper

The simple gripper domain was designed to test another kind of added path complexity. This domain is the simple hole in the wall (parameterized by x, y, and θ), but the robot has different kinematics and must ultimately finish by "grasping" a block. We again chose a tight spot segment length of 15.

In this domain, we find that more training data is required to achieve substantial speedup over the unmodified planner; figure 8 shows the performance as the amount of training data is increased. Even with 500 training points, however, the suggestions allow the planner to find a solution in 97.5% of the problem for which either method succeeds, compared to about 92% for the unmodified planner. As the training data increases, the success rate climbs above 99%.

Figure 7 shows example path segments from different strategies in different environments. The path segments in figure 7 are in roughly the right part of the configuration space, but they tend to involve collisions. The colliding configurations are removed, and so the suggested path segments require patching by the planner, they are less beneficial to the planner than a completely connected path segment would have been.

V. Conclusions

The simple experiments described in this paper show that it is possible to considerably speed up planning time, given experience in related problems, by describing the problems in terms of an appropriate parametric representation and then

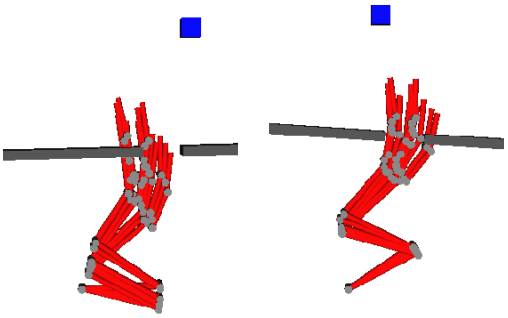

Fig. 7. Suggested path segments for the simple gripper in a single environment.

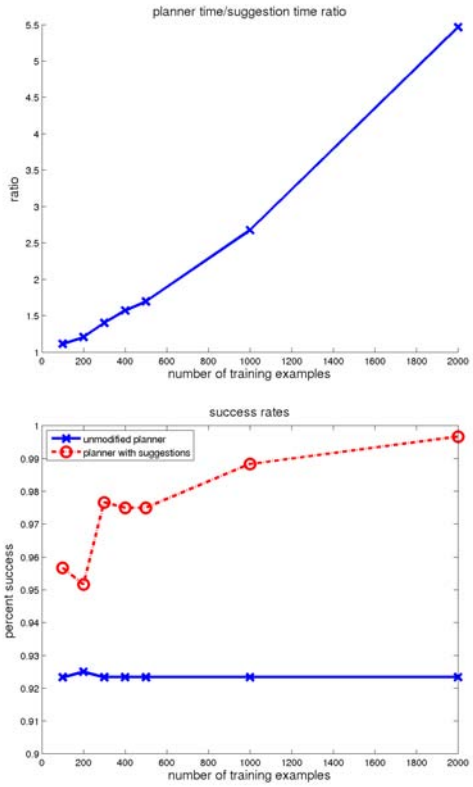

Fig. 8. Results for the simple gripper as the size of the training set is increased. The top plot shows the ratio of the unmodified average planning time to the average planning time with suggestions. The bottom plot shows the percentage of the time that each method finds a plan successfully.

learning to suggest path sub-sequences that are in the most constrained parts of the space.

To improve the performance of this method, we will need to develop a more efficient way to extract and align the sub-sequences of paths from the training examples. In addition, it would broaden the applicability of this method if we could automatically derive task-template parameters from environment descriptions and, ultimately, learn a small set of generally useful task templates from a body of experience in a variety of environments. Lastly, the issue of how to reduce the data requirements in the more complicated domains must be addressed.

Ultimately, this suggests a different direction for path-planning: systems that can routinely and quickly identify the hard spots in any new planning instance and then fill in the rest of the plan, dramatically improving their performance from experience.

Acknowledgment

This research was supported in part by DARPA IPTO Contract FA8750-05-2-0249, "Effective Bayesian Transfer Learning"

References

[1] B. Burns and O. Brock. Sampling-based motion planning using predictive models. *Proceedings of the 2005 IEEE International Conference on Robotics and Automation (ICRA 2005)*, pages 3120–3125, 2005.

[2] B. Burns and O. Brock. Single-query entropy-guided path planning. *Proceedings of the 2005 IEEE International Conference on Robotics and Automation (ICRA 2005)*, 2005.

[3] S. Gaffney and P. Smyth. Trajectory clustering with mixtures of regression models. In *Conference in Knowledge Discovery and Data Mining*, pages 63–72, 1999.

[4] D. Hsu, T. Jiang, J. Reif, and Z.Sun. The bridge test for sampling narrow passages with probabilistic roadmap planners. *EIII/RSJ International Conference on Intelligent Robots and Systems*, pages 4420–4426, 2003.

[5] D. Hsu and H. Kurniawati. Workspace-based connectivity oracle: An adaptive sampling strategy for prm planning. *Proceedings of the International Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2006.

[6] D. Hsu, G. Sánchez-Ante, H l. Cheng, and J-C. Latombe. Multi-level free-space dilation for sampling narrow passages in prm planning. *Proceedings of the 2006 IEEE International Conference on Robotics and Automation (ICRA 2006)*, pages 1255–1260, 2006.

[7] N. M. Amato J-M Lien, S. L. Thomas. A general framework for sampling on the medial axis of the free space. *Proceedings of the 2003 IEEE International Conference on Robotics and Automation (ICRA 2003)*, pages 4439–4444, 2003.

[8] H. Kurniawati and D. Hsu. Workspace importance sampling for probabilistic roadmap planning. *EIII/RSJ International Conference on Intelligent Robots and Systems*, 2004.

[9] M. Morales, L. Tapia, R. Pearce, S. Rodriguez, and N. Amato. A machine learning approach for feature-sensitive motion planning. *Proceedings of the International Workshop on the Algorithmic Foundations of Robotics (WAFR)*, pages 361–376, 2004.

[10] S. Rodriguez, S. Thomas, R. Pearce, and N. Amato. Resampl: A region-sensitive adaptive motion planner. *Proceedings of the International Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2006.

[11] M. Saha and J-C. Latombe. Finding narrow passages with probabilistic roadmaps: The small step retraction method. *EIII/RSJ International Conference on Intelligent Robots and Systems*, 2005.

[12] G. Sanchez and J-C. Latombe. A single-query bi-directional probabilistic roadmap planner with lazy collision checking. In *International Symposium on Robotics Research*, 2001.

[13] T. Siméon, J. Laumond, and C. Nissoux. Visibility-based probabilistic roadmaps. *Proceedings of the IEEE International Conference on Intelligent Robots and Systems*, pages 1316–1321, 1999.

[14] A. F. van der Stappen V. Boor, M. H. Overmars. The gaussian sampling strategy for probabilistic roadmap planners. *Proceedings of the 1999 IEEE International Conference on Robotics and Automation (ICRA 1999)*, pages 1018–1023, 1999.

[15] J. P. van den Berg and M. H. Overmars. Using workspace information as a guide to non-uniform sampling in probabilistic roadmap planners. *The International Journal of Robotics Research*, 24(12):1055–1071, 2005.

[16] Y. Yang and O. Brock. Efficient motion planning based on disassembly. *Proceedings of Robotics: Science and Systems*, 2005.

[17] A. Yershova, L. Jaillet, T. Siméon, and S. LaValle. Dynamic-domain rrts: Efficient exploration by controlling the sampling domain. *Proceedings of the 2005 IEEE International Conference on Robotics and Automation (ICRA 2005)*, 2005.