Randomized Statistical Path Planning

Rosen Diankov James Kuffner

***Abstract*— This paper explores the use of statistical learning methods on randomized path planning algorithms. A continuous, randomized version of A* is presented along with an empirical analysis showing planning time convergence rates in the robotic manipulation domain. The algorithm relies on several heuristics that capture a manipulator's kinematic feasibility and the local environment. A statistical framework is used to learn one of these heuristics from a large amount of training data saving the need to manually tweak parameters every time the problem changes. Using the appropriate formulation, we show that motion primitives can be automatically extracted from the training data in order to boost planning performance. Furthermore, we propose a Randomized Statistical Path Planning (RSPP) paradigm that outlines how a planner using heuristics should take advantage of machine learning algorithms. Planning results are shown for several manipulation problems tested in simulation.**

I. Introduction

In order to solve a specific domain of tasks in path planning or manipulation, most complex robotic systems require prior knowledge of the environment and the robot. Such knowledge usually requires knowing the robot's kinematic capabilities, the type of environment, consequences of traversing configuration paths in the environment, the definitions of a goal condition, and what composes optimal or desired solutions. While a lot of this domain knowledge is necessary to solve the problem, it begs the question of how much the robot can autonomously learn from its domain and how much the human designer needs to specify. An example of knowledge that can be inferred automatically are the reward functions and path heuristics that are very commonly used in planners. Creating these functions is time consuming and it is not trivial to go from a desired robotic behavior to a reward function that encodes this behavior. But given training instances of desired path trajectories from various initial and goal conditions, a robot should be able to learn its own reward and heuristic functions so that it mimics the desired behavior. Another example of domain knowledge is the relationship between configuration space and task space. Usually inverse kinematics are used to compute the goal configuration from the given task, transforming the planning problem from the workspace to the configuration space. The first drawback with IK is that the configuration might not be reachable until the path planning algorithm is executed and fails. The second drawback is that the relationship between the two spaces is highly dependent on the task the robot is supposed to perform. Unless domain specific knowledge is encoded, most robots will act sub-optimally in a complex environment. We focus on solving these problems using statistical learning methods.

Most of planning in continuous domains centers around search based methods or road maps. Search based methods, including randomized search algorithms like RRTs [6], rely on finding a path using heuristics. Most of the time, these heuristics are hand optimized for the particular robot and environment. The other alternative is to create a connected graph in the configuration space so that planning is reduced to graph search combined with local planners [13]. Each planner has its own advantages and disadvantages, and a lot of research in the path planning community deals with proposing heuristics to make up for inherent disadvantages in a planning algorithm [1]. In fact, any type of traditional planner heavily relies on heuristics for fast convergence.

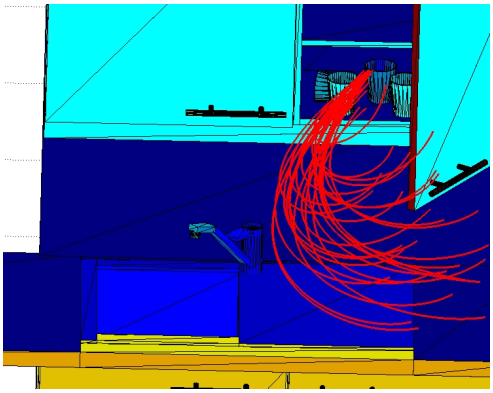


Fig. 1. Predicted end-effector trajectories from the learned motion primitive to take cups out of the cupboard given just the initial and goal positions of the end effector.

Recently, heuristics trained using statistical learning methods have been gaining popularity. [9] learns the direction to approach an object such that the grasping quality is the highest[1]. [3] trains a classifier to compute whether there exists a plan for given starting and goal configurations, addressing the problem that randomized planners never return if a solution does not exist. Creating the right heuristics given a goal behavior of the robot is not a trivial task. Max-Margin Planning [10] addresses this problem in a discrete, low-dimensional domain by borrowing ideas from structured learning [15]. Structured learning is necessary because the transformation from costs to trajectories cannot be easily parameterized. The MMP algorithm parameterizes its cost function as a weighted sum of features extracted from the environment. In order to get the appropriate weights, A* acts as the transformer from costs to trajectories and is used to compute a sub-gradient of the weights with respect

[1]Grasping quality is measured by force closure.

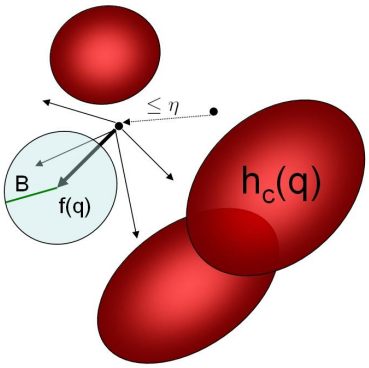


Fig. 2. Graphical representation of all the components in Randomized A*. The red regions represent high cost regions, the arrows represent directions to the child nodes, $f(q)$ is the sampling policy, and the light blue region represent the sampling radius.

to the optimization criteria. One drawback is that Max-Margin Planning requires an optimal planner in the given domain, making it infeasible in high-dimensional domains like manipulation and humanoid locomotion tasks.

First, we propose a path planning algorithm that can return a semi-optimal path with respect to several distance metrics that encode all the domain information needed: a cost function, a distance metric, a goal heuristic, and a sampling-based policy. We call this planner Randomized A* because it is based on A* and the RRT family of algorithms. We use this planner to produce solutions to random manipulation problem instances. At first, solutions are slowly generated off-line to build a training set. In order to build the cost function, we show how motion primitives specific to the task can be automatically extracted from the training data. We define the cost by how far the planner strays from the motion primitive. On the other hand, the policy is trained by analyzing the local environment around the manipulator for keyholes and openings. Because we require training examples, the framework we propose is meant for commonly occurring tasks that need to be solved repeatedly; for example, picking up cups from a sink and putting them in a cupboard. Although simple, the problem can become arbitrarily complex as various obstacles are added to the kitchen. We show that we can learn a generalized model for this task without encoding manipulator specific inverse kinematics or kitchen specific motion primitives.

II. Randomized A*

A* relies on a discrete set of actions to choose from for every discrete state it travels to. Continuous domains have infinitely many states and actions, and finding the correct discretization is usually impossible. RA* lowers the branching factor by randomly sampling the action space for the neighbors to a node. It stores all previously visited nodes in a space-partitioning data structure without binning them[2]. In order to keep nodes from clumping together, the array has the property that a sphere of radius η around any node does not contain other nodes. To compute the neighbors of each node, RA* randomly generates L children around a ball of radius B in the configuration space. If a child can't be reached from its parent or it is *close* to any previously generated node, it will be discarded. Thus, all nodes are guaranteed to have at most L dispersed children. Each child evaluates its total cost like in A*, and the child with the least total cost will be traversed first (see Figure 2). There are a couple of parameters that control the evolution of the search:

- a cost function $h_c : \mathcal{C} \rightarrow \mathbb{R}$ where $\mathcal{C}$ is the configuration space.
- a distance metric $\rho : \mathcal{C} \times \mathcal{C} \rightarrow \mathbb{R}$
- a goal heuristic $h_g : \mathcal{C} \rightarrow \mathbb{R}$
- a sampling radius B for creating children
- a branching factor L (5 is used for all experiments)
- a distance threshold η that controls how close two states need to be to be counted as the same state.

Given a continuous trajectory $\tau(t)$ where $t \in [0, 1]$, we define the cost of the trajectory through configuration space as:

$$\int_0^1 h_c(\tau(t)) \, |\tau'(t)| dt \tag{1}$$

Since RA* returns a discrete path $P = \{c_0, c_1, ..., c_n\}$, we approximate this continuous cost by minimizing

$$f(P) = \sum_{i=1}^{n} \rho(c_{i-1}, c_i) h_c(c_i). \tag{2}$$

Note the subtle differences between A* where the optimization criteria would be to just minimize $f_{A*}(P) = \sum_{i=1}^{n} h_c(c_i)$. This difference is because A* usually has a discrete set of uniformly distributed states, so the distance between each state is roughly the same and can be safely ignored. In RA*, the distances are random and depend on the sampling radius B. Therefore, a normalization factor is needed to make sure that a finely sampled path has roughly the same cost as the same path with fewer waypoints. If the cost function is highly nonlinear, this obviously won't hold for really big distances; but because the sampling radius is bound by B, Equation 2 can be treated as a linearization of the true cost in Equation 1.

Let $c' = Sample(c, B)$ return a random sample c' such that $\rho(c, c') < B$. See Algorithm 1 for an implementation of Randomized A*.

A. Analysis

One advantage with Randomized A* is that it does not need the goal explicitly in configuration space. Most real world manipulators are redundant, so there are many solutions for a given grasping or moving problem. When using IK, the question turns to which solution to pick; however, this problem is not present in Randomized A*.

We tested both a goal directed version of RRTs and Randomized A* on a simple block pickup problem. If the end effector is within an ϵ to its goal position, the planner will return a solution. Even after 30,000 node expansions, RRTs were not able to converge to a solution while Randomized

[2]We use kd-trees for the RA* implementation

Algorithm 1: Randomized-A*

Input: initial configuration c_{init}
// $Insert$ both into $List$ and $ListAll$
$Insert(c_{init}$, null, 0, 0) // no parents, f = 0, g = 0
$c_{best} \leftarrow$ **null**
while *True* **do**
    // get the best node such that f is the lowest.
    // here $g = h_g(c)$
    {c, f, g} $\leftarrow$ List.pop_best()
    if $g < \epsilon$ **then**
        $c_{best} \leftarrow c$
        break
    $children \leftarrow 0$
    for *iter = 1 to MaxIter* **do**
        c' = Sample(c, B)
        $\{c_{near}, f_{near}, g_{near}\} \leftarrow ListAll.nearest(c')$
        if not *InCollision(c')* **and** $\rho(c_{near}, c') > \eta$ **then**
            $cost_{new} \leftarrow f_{near} - g_{near} + h_c(c')\rho(c_{near}, c')$
            Insert(c', c, $cost_{new} + h_g(c')$, $h_g(c')$)
            $children \leftarrow children + 1$
            if $children = L$ **then**
                break
    end
end
Extract Path by back-tracing from c_{best}

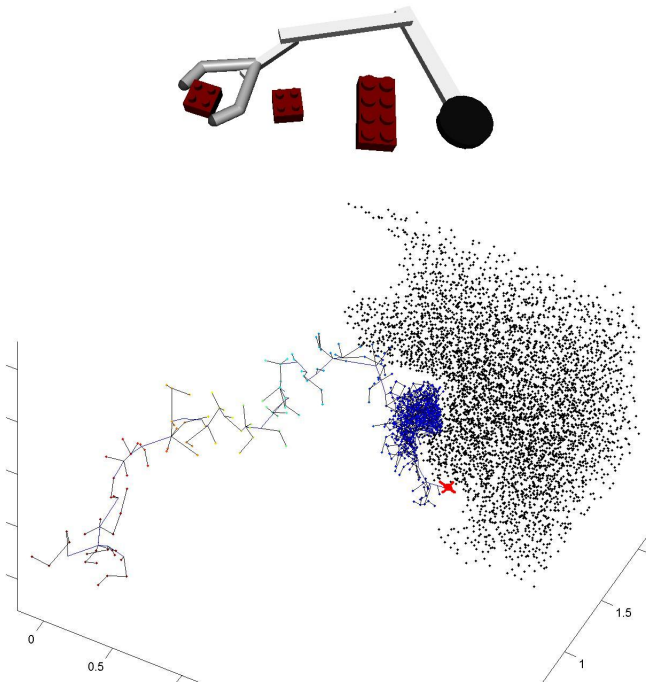


Fig. 3. The node expansion graph of RA* of a 3 degree of freedom planar robot arm. The initial configuration is at the lower left corner and the goal is the big red mark. While most runs were able to converge fast, this particular run got stuck in a cusp created by the collision objects. After a couple hundred iterations, the nodes crowded the space and got through to the goal.

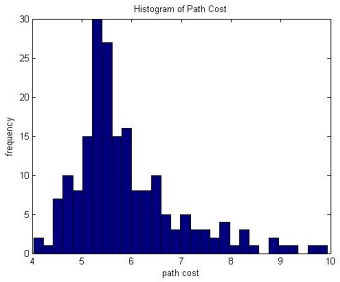


Fig. 4. Histogram of path lengths of a fixed problem over many runs. As can be seen, RA* has a loose bound on optimality, but most paths are still close to the minimum.

A* is able to find solutions close to 100% of the time. This can be attributed to two reasons. First, Randomized A* explores only promising space while regular RRTs explore the whole space. Figure 3 shows the space exploration for a toy problem. Note that space exploration only occurs in the regions closest to the goal. RRT Connect [6], a faster RRT algorithm, could not be used because the goal configuration isn't known *apriori*.

Because a lot of precision is needed to hit a small ball of radius ϵ in configuration space, usually Randomized A* will clump around that region until it generates a sample inside it. This can be avoided by making the goal threshold ϵ bigger. After many trials, we conclude that it is better to increase ϵ a little and use the Jacobian to converge on the closest solution[3]. We use the Damped Least Squares [8] method for the joint update:

$$\delta\theta = J^T(JJ^T + \lambda^2 I)^{-1}(\mathbf{e}_{final} - \mathbf{e}_{initial}) \qquad (3)$$

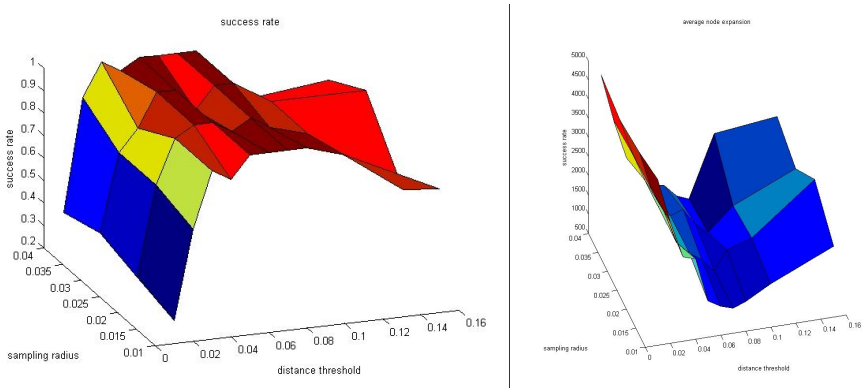
where $J = \frac{\delta \mathbf{e}}{\delta \theta}$ is the Jacobian, $\mathbf{e}$ is the end effector position, and λ is a regularization parameter.

The question of optimality always arises when talking about A*. Figure 4 shows a histogram of path costs from various runs of the same problem. Most solutions do come close to the optimal path. After producing a solution with randomized planners, usually path smoothing is done to the raw solution to minimize it within its homotopy class. Here, it is important that the randomized planner at least hits the correct homotopy class.

III. Learning Heuristics

When first using RA*, it is important to calibrate the ratio between h_c and h_g, just like in A*. If h_c is big compared to h_g, then RA* will have no incentive to explore and will most likely start a breadth-first search in state space. If h_c is small compared to h_g, then the algorithm can get stuck in local minima more easily, the returned solution will also be less optimal. Once an appropriate ratio is found, use it to create the training data regardless of optimality or convergence rates. In general, the best way to obtain the ratio is by testing various values on random problem instances.

[3]Since ϵ is still small, gradient descent does not have any local minima.



(a) Success Rate (b) Average Nodes

Fig. 5. Shows the dependence of RA* on the distance threshold and sampling radius parameters. Average Nodes are directly proportional to the time of the algorithm.

A. Cost Function

The cost function penalizes the robot for being in a certain state. It can be used to guide the robot around regions of singularity/instability and for getting solutions faster because the robot, in some sense, greedily descends using the cost function when searching. Furthermore, the goal is to learn a cost function such that the optimal solutions that Randomized A* produces match the training data as much as possible. The training data itself is produced off-line by gathering data from a human operator controlling the robot, or by running Randomized A* overnight.

The goal is to look for patterns in each trajectory [4] $\tau_i(t)$. Imagine that an arm takes objects from a sink and places them on a counter (see Figure 7). One easily visible pattern to all trajectories is that the end effector goes up first to get out of the sink. Then its height remains roughly constant until right under the cup's goal position (Figure 6). The goal is to extract a motion primitive $\mu(t)$ to capture this behavior. μ is the trajectory of the end-effector in the workspace, and has two constraints at its ends:

$$\mu(0) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad \mu(1) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{4}$$

The end values are chosen arbitrarily. Once $\mu(t)$ is found, we use it to compute the real motion primitive path $\mu'(t)$ the manipulator should follow in the same way as [4] by transforming it so that the start and goal positions match the workspace. Given the real initial position $\mathbf{s}$ and the final goal workspace position $\mathbf{g}$, we compute an affine transformation $A = [R|T]$ such that

$$\mu'(t) = A \circ \mu(t) = R\mu(t) + T \tag{5}$$

$$\mu'(0) = \mathbf{s} \qquad \mu'(1) = \mathbf{g} \tag{6}$$

Note that there is still one degree of freedom: the roll around the line going through s and g. There are a couple of ways to deal with this. One is to sample the roll until a trajectory away from obstacles is found[5]. Another is to statistically measure the trajectory roll from the training data and apply that. Figure 6 shows the final trajectories calculated from the motion primitives and initial and goal positions by measuring the roll statistically.

In this paper, we parameterize μ by a cubic function. Cubic functions in workspace are enough to capture simple patterns in a kitchen environment. However, the following framework applies to any parameterization of motion primitives.

The cost function is built by penalizing areas far away from the final path μ'. Solving for the minimum distance for a cubic polynomial involves finding the roots of a 5^{th} order polynomial, a numeric calculation. A quicker, more robust way to compute cost is to sample μ' at a discrete number of time steps $\{t_i\}$. The minimum distance can be approximated by finding the closest sample on the motion primitive to a given point. Therefore, the cost function is simplified to

$$h_c(q) = \min_i \|ForwardKinematics(q) - \mu'(t_i)\|^2 \tag{7}$$

Working with workspace trajectories allows generalization across different manipualtors, and the trajectory is always in 3D. The motion primitives themselves are designed to fit the environment, so a workspace motion primitive for the end-effector makes more sense than a configuration space motion primitive. Table I shows results using motion primitives for various problems.

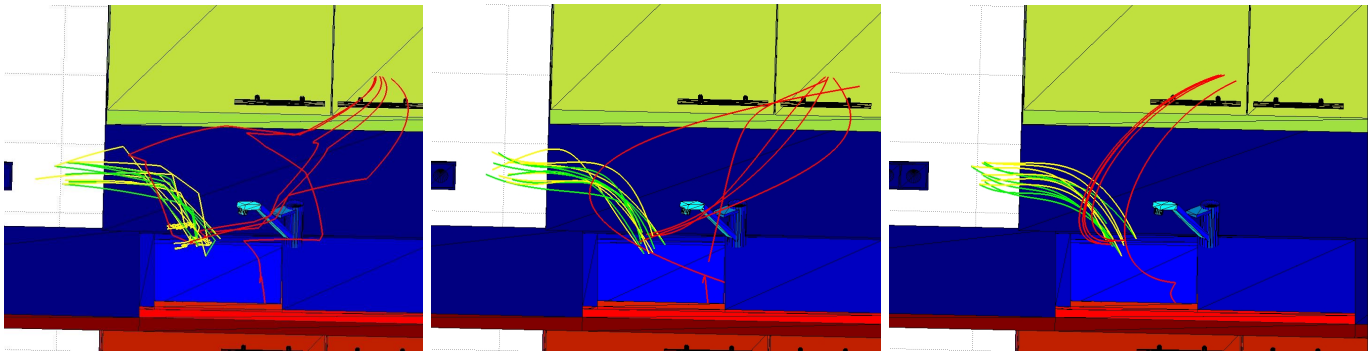
B. Extracting Motion Primitives

In order to remove noise, a cubic polynomial is fit to all training trajectories $\tau_i(t)$,

$$\tau_i'(t) = C_i \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix} \tag{8}$$

[4]All trajectories are defined by a piecewise linear function of waypoints and $t \in [0, 1]$.

[5]The trajectories computed from motion primitives don't necessarily have to be collision free.



(a) (b) (c)

Fig. 6. Constructing motion primitives from training data. Each trajectory is the position of the end-effector through time. (a) Shows trajectories of many runs. Red is for coming to the sink from the top. Yellow is for bringing the glass from the sink to the counter. Green is going from the counter to the sink. (b) Trajectories with cubic functions fit to them. (c) Predicted trajectories using the computed motion primitives. One motion primitive is computed for every color.

such that τ_i' best approximates τ_i. Before a pattern can be extracted all τ_i' have to be aligned together. The best affine transformation $A_i = [R_i|T_i]$ must be found for each trajectory such that:

$$\sum_{i,j} \|A_i \circ \tau_\mathbf{i} - A_j \circ \tau_\mathbf{j}\|^2 \tag{9}$$

is minimized. It is not very clear how to minimize this since the distance between two cubic functions itself is not a simple expression. Therefore, assume that in the best alignment, for any $t_0 \in [0, 1]$

$$\forall_{i,j}\ \tau_i(t_0) = \tau_j(t_0) \tag{10}$$

Now sample N times from $[0, 1]$ to form a set $\{t_i\}$ and define $X_{i,j} = \tau_i(t_j)$. Equation 9 is simplified by minimizing

$$\sum_{i,j} \sum_{k=1}^{N} \|R_i X_{i,k} - R_j X_{j,k} + (T_i - T_j)\|^2. \tag{11}$$

We compute the alignment by using Generalized Procrustes Analysis [14] (Figure 6). Once aligned, we extract the mean cubic polynomial and transform it such that Equation 4 holds. Note that alignment and motion primitive extraction could be done using more general splines.

C. Paradigm

The driving force behind learning in path planning is that researchers optimize heuristics for their specific problems all the time in order to ensure good convergence rates and solutions. Cognitive research has good evidence that humans create motion primitives and other heuristics on the fly as specific problems are encountered [11]. The question is how to create path planners that embed such a learning framework automatically. In this paper, we diverge a little from the mainstream path planning methods in that we don't seek to create a general heuristic to path planners that can be applied to all problems. Instead, we seek to learn heuristics specific to the problem at hand. These heuristics are data-driven in that they require previous attempts of solving the problem in order to encode domain specific knowledge. It should be mentioned here that a learned heuristic should still generalize well across different problem instances of the same family of problems.

	Success Rate	Node Expansion($\propto$ time)
Cup in Sink	96%	1620
(with Motion Primitives)	**100 %**	**1397**
Cluttered Cup in Sink	76%	2500
(with Motion Primitives)	**90%**	**1737**
Cluttered Cup in Cupboard	90%	2334
(with Motion Primitives)	**97%**	**2053**

TABLE I

TABLE SHOWS A COMPARISON OF MOTION PRIMITIVES AVERAGED OVER MANY RUNS OF SEVERAL PROBLEMS. IN EACH PROBLEM, THE START AND GOAL POSITION FOR THE ROBOT CHANGES. THE NON MOTION PRIMITIVE COST WAS CHOSEN TO BE A CONSTANT VALUE THAT GIVES THE BEST RESULTS. RUNS EXCEEDING 5000 RA* NODE EXPANSIONS ARE DECLARED TO BE A FAILURE.

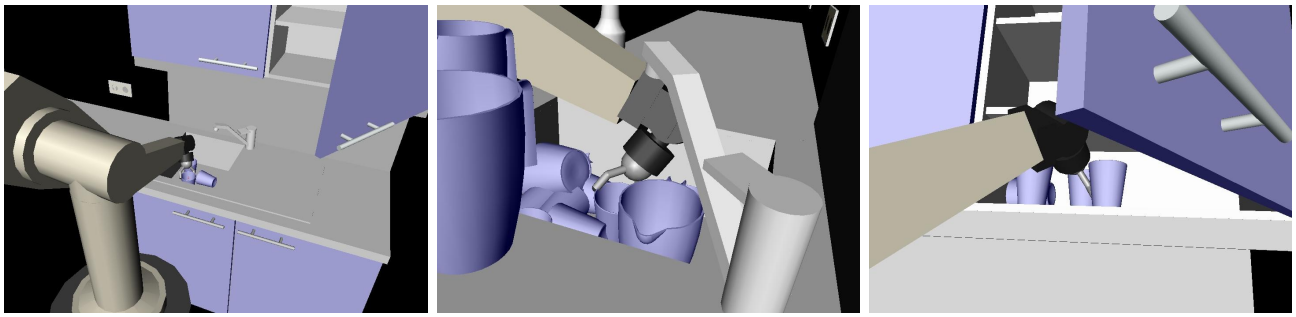
Regardless of the heuristics, the path planning algorithm used is still vital to success. The main reason a randomized version of A* is proposed is that it is heuristic driven, works in continuous spaces, and is randomized. Randomization is necessary to ensure probabilistic completeness and a certain degree of spontaneity in the system [7]. Note that each heuristic manages its own space of important factors:

- **cost function** - Environment specific
- **distance metric** - Robot specific, encodes how far configurations of the robot are with respect to each other.
- **goal function** - Goal specific, can be arbitrary.
- **sampling function** - Local policy specific.

Combined, the heuristics can summarize a good amount of domain specific information.

IV. Results and Conclusion

This paper proposes and analyzes several ways to deal with planning heuristics in continuous spaces by using learning. A randomized version of A* is presented that converges relatively fast in high-dimensional spaces. All tests were done with the PUMA robot arm performing several tasks



(Cup in Sink) (Cluttered Cup in Sink) (Cluttered Cup in Cupboard)

Fig. 7. The various test runs used to evaluate RA* and motion primitives.

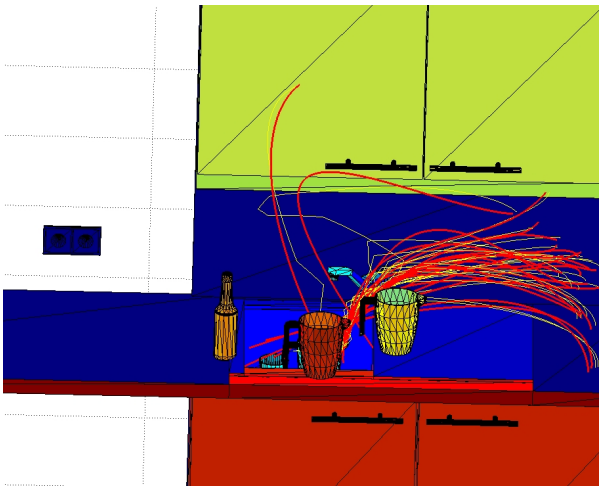


Fig. 8. Another comparison of training data (yellow) and the trajectories μ' (red) constructed from the motion pritive using only the initial start and goal positions in workspace.

in a cluttered kitchen environment. Because Randomized A* does not rely on the explicit inverse kinematics of the robot, it is applicable to any type of robot. The algorithm is very similar to discrete A* except that it requires two more parameters to deal with neighbors in continuous space: the distance threshold and the sampling radius.

To enforce RA*, a data-driven heuristic is presented. It extracts and uses motion primitives from analyzing trajectories of the manipulator in workspace coordinates. The motion primitives are general enough to be used again on similar problems, but specific enough to recognize patterns (see Figure 8). The fact that primitives are generated automatically means that a robot can have a motion primitive for any family of tasks it is given.

A. *Future Work*

Although we use RA* only for manipulation planning, it could be applied in other areas like in non-holonomic planning to find and reuse local policies. One goal is to apply RSPP to humanoid navigation on rough terrain. It is still a big question whether useful motion primitives could be automatically extracted from humanoids given large amounts of data.

V. Acknowledgements

This material is based upon work supported in part by the National Science Foundation under grant EEC-0540865.

References

[1] D. Bertram, J.J. Kuffner, T. Asfour, and R. Dillman. *A unified approach to inverse kinematics and path planning for redundant manipulators*. In Proc. IEEE Int. Conf. on Robotics and Automation (ICRA'06), pages 1874-1879, 2006.

[2] T. F. Cootes, G. J. Edwards, and C. J. Taylor. *Active appearance models*. IEEE TPAMI, 23(6):681685, 2001.

[3] K. Hauser, T. Bretl, J. Latombe. *Learning-Assisted Multi-Step Planning*. In Proc. IEEE Int. Conf. on Robotics and Automation (ICRA'05), April 2005.

[4] K. Hauser, T. Bretl, J. Latombe. *Using Motion Primitives in Probabilistic Sample-Based Motion Planning for Humanoid Robots*. Workshop on the Algorithmic Foundations of Robotics (WAFR), 2006.

[5] M. Kallmann, M. Mataric'. *Motion Planning Using Dynamic Roadmaps*. In Proc. IEEE Int. Conf. on Robotics and Automation (ICRA'04), April 2004.

[6] J.J. Kuffner and S.M. LaValle. *RRT-Connect: An efficient approach to single-query path planning*. In Proc. IEEE Int'l Conf. on Robotics and Automation (ICRA'2000), pages 995-1001, San Francisco, CA, April 2000.

[7] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, New York, NY, 2006

[8] Y. Nakamura, H. Hanafusa. *Inverse Kinematics solutions with Singularity Robustness for Robot Manipulator Control*. Journal of Dynamic Systems, Measurement, and Control. 108 (1986), pp. 163-171.

[9] R Pelessof, A. Miller, P. Allen, T. Jebara. *An SVM Learning Approach to Robotic Grasping*. In Proc. IEEE Int. Conf. on Robotics and Automation (ICRA'04), April 2004.

[10] Nathan Ratliff, J. Andrew Bagnell, Margin Zinkevich. *Maximum Margin Planning*. Proceedings of the 23^{rd} International Conference on Machine Learning, 2006.

[11] Stefan Schaal. *Is imitation learning the route to humanoid robots?*. Trends in Cognitive Sciences, Vol 3, No 6, June 1999.

[12] Matthias Seeger. *Gaussian Processes for Machine Learning*. International Journal of Neural Systems. Vol 14, issue 2, 2004.

[13] T. Simon, J.-P. Laumond, J. Corts, and A. Sahbani. *Manipulation planning with probabilistic roadmaps*. International Journal of Robotics Research, vol. 23, no. 7, pp. 729746, 2004.

[14] Mikkel B. Stegmann, David Delgado Gomez. *A Brief Introduction to Statistical Shape Analysis*. Technical University of Denmark, Lyngby, 2002.

[15] Ben Taskar, Vassil Chatalbashev, Daphne Koller, Carlos Guestrin. *Learning Structured Prediction Models: A Large Margin Approach*. Proceedings of the 22^{nd} International Conference on Machine Learning, 2005.