Learning an Interface to Improve Efficiency in Combined Task and Motion Planning

Rohan Chitnis[1], Dylan Hadfield-Menell[2], Siddharth Srivastava[3], Abhishek Gupta[2], and Pieter Abbeel[2]

***Abstract*— In mobile manipulation planning, it is not uncommon for tasks to require thousands of individual motions. Planning complexity is exponential in the length of the plan, rendering direct motion planning intractable for many problems of interest. Recent work has focused on *task and motion planning* (TAMP) as a way to address this challenge. TAMP methods integrate logical search with continuous geometric reasoning in order to sequence several short-horizon motion plans that together solve a long-horizon task. To account for continuous parameters, many of these systems rely on hand-coded discretizations of the domain. Such an approach lacks robustness and requires substantial design effort. In this paper, we present methods to improve the reliability and speed of planning in a TAMP system. The approach we build on first plans abstractly, ignoring continuous values, and then performs *plan refinement* to determine feasible parameter settings. We formulate plan refinement as a Markov decision process (MDP) and give a reinforcement learning (RL) algorithm to learn a policy for it. We also present initial work that learns which plan, from a set of potential candidates, to try to refine. Our contributions are as follows: 1) we present a randomized local search algorithm for plan refinement that is easily formulated as an MDP; 2) we give an RL algorithm that learns a policy for this MDP; 3) we present a method that trains heuristics for selecting which plan to try to refine; and 4) we perform experiments to evaluate the performance of our system in a variety of simulated domains. We show improvements in success rate and planning time over a hand-coded baseline.**

I. Introduction

A long-term goal of robotics research is the introduction of intelligent household robots. To be effective, such robots will need to perform complex tasks over long horizons (e.g., setting a dinner table, doing laundry). Planning for these long-horizon tasks is infeasible for state-of-the-art motion planners, making the need for a hierarchical system of reasoning apparent.

One way to approach hierarchical planning is through combined *task and motion planning* (TAMP). In this approach, an agent is given a symbolic, logical characterization of actions (e.g., move, grasp, putdown), along with a geometric encoding of the environment. TAMP systems maintain a hierarchical separation of high-level, symbolic task planning and low-level, geometric motion planning. Efficient integration of these two types of reasoning is challenging, and recent research has proposed several methods for it [1], [2], [3], [4], [5].

Our methods build on the TAMP system presented by Srivastava et al. [1]; similar techniques could be applied for the TAMP approaches cited above. In Srivastava et al., a (classical) task planner produces a symbolic plan containing a sequence of actions to reach a goal state. Then, in a process known as *plan refinement*, candidate values are proposed for the continuous variables in this symbolic plan. These values are checked locally for feasibility by calling a motion planner. An advantage of this system is that the task planner and the motion planner are used as black boxes.

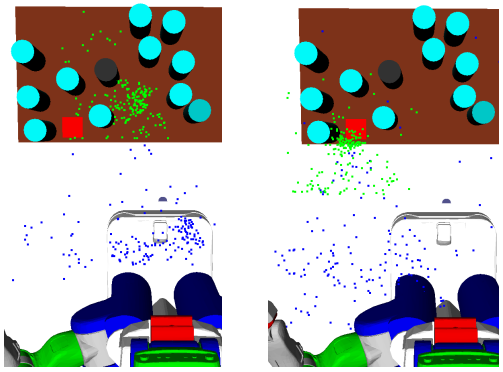


Fig. 1: Screenshots showing distributions learned by our system in a simulated pick-and-place domain. We use reinforcement learning to train good sampling distributions for continuous motion planning parameters in long-horizon tasks. The robot must grasp the black can and put it down on the red square. The left image shows learned base position (blue) and grasping (green) distributions, and the right shows learned base position (blue) and putdown (green) distributions. The grasping policy learned to avoid the obstructions.

The authors propose an *interface layer* for refining the plan into a set of collision-free trajectories; it performs an exhaustive backtracking search over a set of sampled candidate parameter values, which are instantiations of symbolic references in the plan. If a motion planning feasible refinement is not found within the resource limit, symbolic error information is propagated back to the task planner, and a new symbolic plan is produced.

The TAMP paradigms referenced above all use hand-coded heuristics to instantiate symbols with continuous values. The system presented by Srivastava et al. relies on hand-coded discretizations to sample values for plan parameters. Designing these heuristic sampling distributions often requires substantial effort. They typically depend on geometric attributes of the environment and its objects and must be re-calibrated to run the system in a new setting. Further, the discretizations must be fairly coarse to allow reasonable search speeds, meaning they inherently lack robustness to increased environmental complexity.

Reinforcement learning (RL) refers to the process of an

[1] ronuchit@berkeley.edu
[2] {dhm, pabbeel, abhigupta}@eecs.berkeley.edu
[3] siddharth.srivastava@utrc.utc.com

agent learning a policy (a mapping from states to actions) in its environment that maximizes rewards. Zhang and Dietterich [6] first applied the RL framework to planning problems, using a job shop scheduling setting. In this work, we take inspiration from their approach; we use RL to train continuous proposal distributions for plan refinement in a TAMP system. We implement our approach using methods adapted from Zucker et al. [7], who train a configuration space sampler for motion planning using features of the discretized workspace.

Unpublished work that is currently in progress extends the system in [1] to achieve completeness by maintaining a *plan refinement graph*, whose nodes each store a valid symbolic plan (that reaches the goal) and its current parameter values. This makes it possible to interleave partial refinement of several plans. In this paper, we also develop a method for making the decision of which plan to try refining next.

The four contributions of our work are as follows: 1) we present randomized refinement, a local search algorithm for plan refinement that is easily formulated as an MDP; 2) we formulate plan refinement in the RL framework and learn a policy for this MDP; 3) we train heuristics to search intelligently through a plan refinement graph, allowing us to decide which plan to try refining next; and 4) we present experiments to evaluate our approach in a variety of simulated domains. Our results demonstrate that our approach yields significantly improved performance over that of hand-coded discretizations of the plan refinement sample space.

II. Related Work

Our work uses RL techniques to improve planning reliability and speed in a TAMP system.

Kaelbling et al. [2] use hand-coded "geometric suggesters" to propose continuous geometric values for the plan parameters. These suggesters are heuristic computations which map information about the robot type and geometric operators to a restricted set of values to sample for each plan parameter. Our methods could be adapted here to learn these suggesters.

Lagriffoul et al. [3] propose a set of geometric constraints involving the kinematics and sizes of the specific objects of interest in the environment. These constraints then define a feasible region from which to search for geometric instantiations of plan parameters. Our approach could be adapted to learn generalized versions of these constraints that apply to various domains.

Garrett et al. [4] use information about reachability in the robot configuration space and symbolic state space to construct a *relaxed plan graph* that guides motion planning queries, using geometric biases to break ties among states with the same heuristic value. By contrast, we allow RL to shape distributions for motion planning queries, so reachability information is naturally incorporated.

III. Background

We provide relevant technical background and introduce notation used throughout the paper.

A. Markov Decision Processes and Reinforcement Learning

Markov decision processes (MDPs) are the standard AI approach for formulating interactions between agents and environments. At each step of an MDP, the agent knows its current state and selects an action. This causes the state to change according to a known transition distribution. We define an MDP as a tuple $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma, \mathcal{P} \rangle$, where

- $\mathcal{S}$ is the state space.
- $\mathcal{A}$ is the action space.
- $T(s, a, s') = Pr(s'|s, a)$ for $s, s' \in \mathcal{S}, a \in \mathcal{A}$ is the transition distribution.
- $R(s, a, s')$ for $s, s' \in \mathcal{S}, a \in \mathcal{A}$ is the reward function.
- $\gamma \in [0, 1]$ is the discount factor.
- $\mathcal{P}$ is the initial state distribution.

A solution to an MDP is a policy, $\pi : \mathcal{S} \rightarrow \mathcal{A}$, that maps states to actions. The value, $V_\pi(s)$, of a state under π is the sum of expected discounted future rewards from starting in state s and selecting actions according to π:

$$V_\pi(s) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t R(s_t) | \pi, s_0 = s].$$

The optimal policy, π^*, maximizes this value for all states.

In reinforcement learning (RL), an agent must determine π^* through interaction with its environment (i.e. without explicit access to $\mathcal{S}$ or T). At each timestep, the agent knows the state and what actions are available, but initially does not know how taking actions will affect the state. There is a large body of research on RL, and standard techniques include value function approximation, which uses methods such as temporal difference (TD) learning, and direct policy estimation, which encompasses both gradient-based and gradient-free methods [8].

B. Reinforcement Learning for Planning

Our problem formulation is motivated by Zhang and Dietterich's application of RL to job shop scheduling [6]. Job shop scheduling is a combinatorial optimization problem where the goal is to find a minimum-duration schedule of a set of jobs with temporal and resource constraints. An empirically successful approach to this problem relies on a randomized local search that proposes changes to an existing suboptimal schedule. The authors formulate this as an MDP and use $TD(\lambda)$ [8] with function approximation to learn a value function for it. Their approach outperforms the previous state of the art for this task and scales better to larger scheduling problems.

Zucker et al. [7] use RL to bias the distribution of a rapidly exploring random tree (RRT) for motion planning. Their approach uses features of a discretization of the workspace to train a non-uniform configuration space sampler using policy gradient algorithms. In our work, we adopt their gradient updates for the TAMP framework (Section V-C).

C. Task and Motion Planning

We define task and motion planning (TAMP) as a tuple $\langle \mathcal{O}, \mathcal{T}, \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{C} \rangle$:

- $\mathcal{O}$ is a set of symbolic references to *objects* in the environment, such as cylinders and grasping poses.
- $\mathcal{T}$ is the set of object *types*, such as robot manipulators, cylinders, poses, and locations.
- $\mathcal{F}$ is a set of *fluents*, which define relationships among objects and hold either true or false in the state.
- $\mathcal{I}$ is the set of fluents that hold true in the initial state.
- $\mathcal{G}$ is the set of fluents defining the goal state.
- $\mathcal{C}$ is a set of parametrized *high-level actions*, defined by *preconditions*, a set of fluent literals that must hold true in the current state to be able to perform the action, and *effects*, a set of fluent literals that hold true after the action is performed.

A solution to a task planning problem is a valid sequence of actions $a_0, a_1, ..., a_n \in \mathcal{C}$ which, when applied successively to states starting with $\mathcal{I}$, results in a state in which all fluents of $\mathcal{G}$ hold true.

Classical task planners operate on discrete, purely logical levels and thus cannot directly solve this problem. We adopt the hierarchical method of Srivastava et al. [1] to account for this discrepancy. A key step in their approach is the abstraction of continuous state variables, such as robot grasping poses for objects, into a discrete set of symbolic references to potential values. A *symbolic*, or *high-level*, plan refers to the fixed task sequence returned by a task planner and comprised of these symbolic references. An *interface layer* assigns continuous values to symbolic references, in a process called *plan refinement*. These plan parameters – symbolic references in the plan that are instantiated with continuous values – define a sequence of states. A motion planner is used to find trajectories between these states.

If no satisfying set of parameters can be found (e.g., because an object is obstructed), the interface layer augments the symbolic fluent state to reflect this. Using pose references, this can be done without adding new discrete pose values to the high-level representation. This is essential, because the size of the high-level state space is exponential in the number of discrete values represented at the high level. The system then obtains a new plan from the task planner.

The authors introduce an algorithm for plan refinement called TRYREFINE, an exhaustive backtracking routine over a discrete set of candidate plan parameter values. (Interested readers are referred to [1] for details.) This hand-coded discretization is domain-specific and tuned to the geometry of objects in the environment. For example, in a pick-and-place task, end effector grasp poses for a can are sampled around the can in each of the 8 cardinal and intermediate directions, at a fixed distance and height. Such coarse sampling distributions necessitate fine-tuning for each domain and are not robust to increased environmental complexity.

An unpublished extension to this system that is currently in progress develops a complete algorithm and maintains a *plan refinement graph*, whose nodes each store a valid symbolic plan and its current set of instantiated parameter values. If refinement of a plan p stored in node n leads to discovered facts being propagated to the task planner, the newly produced plan p' based on the updated fluent state is added as a child node n' of n. This makes it possible to divide computational effort between the search for additional high-level plans that resolve specific errors in previous ones, and the search for error-free refinements of existing plans.

Algorithm 1 Randomized refinement.

```
1:  procedure RANDREF(HLP, N)
2:      init ← INITREFINEMENT(HLP)
3:      for iter = 0, 1, ..., N do
4:          failStep, failPred ←MOTIONPLAN(HLP)
5:          if failStep is null then
6:              # Found successful plan refinement.
7:              return success
8:          end if
9:          if failPred is null then
10:             # Motion planning failure.
11:             failAction ← HLP.ops[failStep]
12:             RESAMPLE(failAction.params)
13:         else
14:             # Action precondition violation.
15:             RESAMPLE(failPred.params)
16:         end if
17:     end for
18:     Raise failure to task planner, receive new plan.
19: end procedure
```

IV. RANDOMIZED REFINEMENT

Before we can apply RL to plan refinement, we must formulate it as an MDP. We design our approach to imitate that of Zhang and Dietterich [6]: we initialize an infeasible refinement and use a randomized local search to propose improvements. We maintain a value for each high-level plan variable. At each iteration, a variable whose current value is leading to a motion planning failure or action precondition violation is picked randomly and resampled. Algorithm 1 shows pseudocode for this randomized refinement.

The procedure takes two arguments, a high-level plan and a maximum iteration count. In line 2, we initialize all variables in the high-level plan by sampling from their corresponding distributions, which can be hand-coded or learned, as described in Section V. We continue sampling until we find bindings for symbolic pose references that satisfy inverse kinematics constraints (IK feasibility). Trajectories are initialized as straight lines.

We call the MOTIONPLAN subroutine in line 4, which attempts to find a collision-free set of trajectories linking all pose instantiations. To do so, it iterates through the sequence of actions that comprise the high-level plan. For each, it first calls the motion planner to find a trajectory linking the sampled poses. If this succeeds, it tests the action preconditions; as part of this step, it checks that the trajectory is collision-free.

If MOTIONPLAN is unsuccessful, it returns into $failStep$ the index of the action where failure occurred. If this was due to a motion planning failure, $failPred$ is null. Otherwise, the violated action precondition is stored into $failPred$.

We call the RESAMPLE routine on the high-level parameters associated with a failure; this routine picks one of these high-level parameters at random and resamples it from its distribution. If we reach the iteration limit (l.18), we convert the most recent failure information into a symbolic representation, then raise it to the task planner, which will update its fluent state and provide a new high-level plan.

V. LEARNING REFINEMENT DISTRIBUTIONS

In this section, we present our primary contribution: an RL approach that learns a policy for plan refinement.

A. Formulation as Reinforcement Learning Problem

We formulate plan refinement as an MDP as follows:

- A state $s \in \mathcal{S}$ is a tuple (P, r_{cur}, E, n), consisting of the symbolic plan, its current instantiation of values for plan parameters, the geometric encoding of the environment, and a counter for the number of calls to the sampler.
- An action $a \in \mathcal{A}$ is a pair (p, x), where p is the discrete plan parameter to resample and x is the continuous value assigned to p in the new refinement.
- The transition function $T(s, a, s')$ is split up into 3 cases. In all cases, n increases by 1. L refers to the number of samples for one planning problem.
  - Case 1: $n > L$. We sample a new state from $\mathcal{P}$ and reset n to 0.
  - Case 2: the proposed value x is IK infeasible. The state remains the same.
  - Case 3: Otherwise, the value of p is set to x and the motion planner is called.
- The reward function $R(s, a, s')$ provides rewards based on a measure of closeness to a valid plan refinement.
- $\mathcal{P}$ is a distribution over planning problems, encompassing both the task planning problem and the environment.

We restrict our attention to training policies that suggest x for actions in $\mathcal{A}$. We note that randomized refinement provides a fixed policy for selecting p.

Our reward function R explicitly encourages successful plan refinement, providing positive reward linearly interpolated between 0 and 20 based on the fraction of high-level actions whose preconditions are satisfied. Additionally, we give -1 reward every time we sample an IK infeasible pose, to minimize how long the system spends resampling plan variables until obtaining IK feasible samples.

B. Training Process

We learn a policy for this MDP by adapting the method of Zucker et al. [7], which uses a linear combination of features to define a distribution over poses. In our setting, we learn a weight vector θ_p for each parameter *type*, comprised of a pose type and possibly a gripper (e.g., "left gripper grasp pose," "right gripper putdown pose," "base pose"). These weight vectors encode the policy for selecting x for actions in $\mathcal{A}$. This decouples the learned distributions from any single high-level plan and allows generalization across problems.

We develop a feature function $f(s, p, x)$ that maps the current state $s \in \mathcal{S}$, plan parameter p, and sampled value x for p to a feature vector; f defines a policy class for the MDP. Additionally, we define N as the number of planning problems on which to train, and ϵ as the number of samples comprising a training episode.

The training is a natural extension of randomized refinement and progresses as follows. N times, sample from $\mathcal{P}$ to obtain a complete planning problem Π. For each Π, run the randomized refinement algorithm to attempt to find a valid plan refinement, allowing the RESAMPLE routine to be called L times before termination. Select actions according to the θ_p and collect rewards according to R. After every ϵ calls to RESAMPLE, perform a gradient update on the weights.

C. Distribution and Gradient Updates

We adopt the sampling distribution used in Zucker et al. [7] for a parameter p with sample value x, in state $s \in \mathcal{S}$:

$$q(s, p, x) \propto exp(\theta_p^T f(s, p, x)).$$

The authors define the expected reward of an episode ξ:

$$\eta(\theta_p) = \mathbb{E}_q[R(\xi)]$$

and provide an approximation for its gradient:

$$\nabla\eta(\theta_p) \approx \frac{R(\xi)}{\epsilon} \sum_{i=1}^{\epsilon} (f(s, p, x_i) - \mathbb{E}_{q,s}[f]).$$

$R(\xi)$ is the sum over all rewards obtained throughout ξ, and $\mathbb{E}_{q,s}[f]$ is the expected feature vector under q, in state $s \in \mathcal{S}$. The weight vector update is then:

$$\theta_p \leftarrow \theta_p + \alpha\nabla\eta(\theta_p)$$

for appropriate step size α.

We sample x from q using the Metropolis algorithm [9]. Since our distributions are continuous, we cannot easily calculate $\mathbb{E}_q[f]$, so we approximate it by averaging together the feature vectors for several samples from q.

VI. SELECTING A PLAN TO TRY REFINING

The approach presented thus far can be succinctly described as learning *how* to refine a single high-level plan. In this section, we present a method for learning *which* plan to try refining. Section III-C describes the plan refinement graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, which maintains a set of candidate plans and their current refinements. The decision to be made is of the form (v, b), where $v \in \mathcal{V}$ denotes which node to visit next, and b is a Boolean that indicates the action to be performed at v. There are two possible actions: 1) attempt to find a valid refinement for the plan stored in v, or 2) recognize that a valid refinement does not exist for this plan and generate a geometric fact to use for replanning, by allowing collisions when motion planning for the current instantiation of plan parameters.

To select between the potential refinement options, we learn decision tree regressors to answer the following questions about a single node n containing plan p: 1) how many iterations of randomized refinement would be needed to achieve a valid refinement for p (∞ if p has no valid

# Objects	System	% Solved (SD)	Avg MP Time (s)	Avg # MP Calls
2 (dinner)	B	100 (0)	25.6	59.2
2 (dinner)	L	99 (2.0)	26.0	61.6
4 (dinner)	B	92 (0)	45.8	94.5
4 (dinner)	L	99 (1.6)	39.5	96.2
25 (cans)	B	74 (0)	15.1	19.0
25 (cans)	L	84 (5.1)	12.4	12.6
25 (cans)	F	92 (5.8)	10.7	12.0
30 (cans)	B	36 (0)	48.2	35.4
30 (cans)	L	69 (8.0)	21.9	19.0
30 (cans)	F	77 (6.5)	23.1	21.8

TABLE I: Percent solved and standard deviation, along with time spent motion planning and number of calls to the motion planner for the baseline system (B), our learned refinement policies with depth-first search through the plan refinement graph (L), and our learned refinement policies and heuristics for searching the graph (F). Results for L and F are averaged across 10 separately trained sets of weights. Time limit: 300s.

refinement); 2) if we quickly generate a child node n' under n by discovering geometric facts and producing a new plan p', how many iterations would be needed to refine p'? We approach this by learning an estimate of the number of iterations needed to refine *each* action. To obtain an estimate for a full plan, we sum this number across all of the plan's actions. This implicitly assumes that dependencies in plans are, in some way, local; it only makes sense if a plan can be split into subportions with independent refinements. Addressing this will be an important area of future work.

To train the regressors, we fix pre-trained policies for plan refinement and construct datasets for supervised learning as follows. For the first regressor, we run refinement on the root node of the graph over 500 random environments sampled from $\mathcal{P}$, and measure the feature vector and number of iterations until valid refinement (arbitrarily large if no valid refinement exists). For the second regressor, we do the same, but on a single child node spawned from the root node. We then fit standard decision tree regressors to our data.

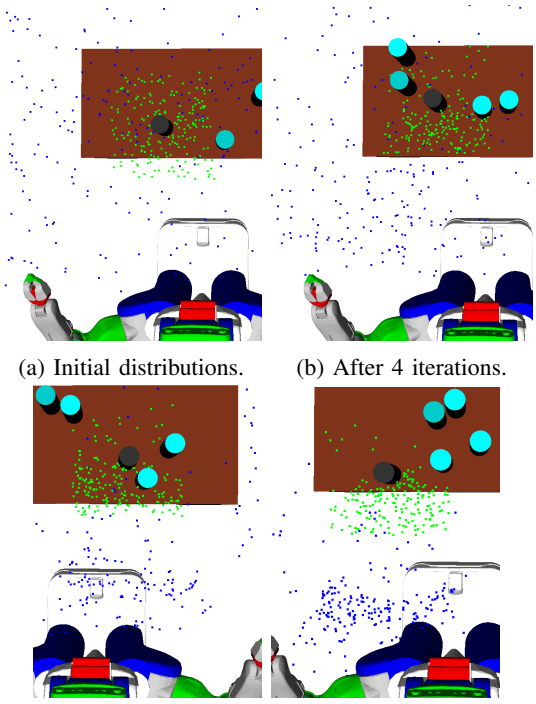
The features for our regressors are as follows. 3 geometric features encode the closeness of the objects of interest in our environment, considering the distance to and placement of nearby obstructions. The other feature describes how many times the node has been visited before.

At test time, we make the decision (v, b) as follows. We select v according to a softmin (with decreasing temperature) over the values predicted by the first regressor. Then, we select b using a softmin comparison between the two regressors' predicted values for v. For example, if refining a child node would reduce the number of steps to a valid refinement, we bias toward selecting action 2.

VII. Experiments

A. Training Methodology

We use the reward function described earlier. Our weight vectors are initialized to $\vec{0}$ for all parameter types – this



(a) Initial distributions. (b) After 4 iterations.

(c) After 8 iterations. (d) Final distributions.

Fig. 2: Learned base position (blue) and left arm grasp (green) distributions used to pick up the black can after different training iterations for learning refinement policies. An iteration refers to a single planning problem, which terminates after L calls to the RESAMPLE routine. The initial distributions are uniform because we initialize weights to $\vec{0}$. The final distributions are after 12 iterations.

initialization represents a uniform distribution across the limits of the geometric search space. We use 24 features for learning the θ_p. 9 binary features encode the bucketed distance between the sample and target (the object referenced by the parameter). 9 binary features encode the bucketed sample height. 3 features describe the number of other objects within discs of radius 7, 10, and 15 centimeters around the sample. 3 binary features describe the angle made between the vector from the robot to the target and the vector from the sample to the target: whether the angle is less than $\pi/3$, $\pi/2$, and $3\pi/4$.

Initial experimentation revealed that training weights for all parameter types jointly is intractable, because planning takes a long time. Potential solutions for this would explore alternative RL algorithms, but this is not our focus. Instead, we apply curriculum learning by training with a planning problem distribution $\mathcal{P}$ that gets progressively harder. Additionally, we train the refinement policies first, then fix them while training the graph search heuristics.

We evaluate our approach in two distinct domains: cans distributed on a table (the *can domain*) and setting up bowls for dinner (the *dinner domain*). We compare performance with a baseline that uses the hand-coded sampling distributions in Srivastava et al. [1] and depth-first search of the plan refinement graph, which always refines the plan that incorporates all error information obtained thus far. For the can domain, we report results for 3 systems: 1) this baseline,

2) our learned refinement policies with the depth-first search, and 3) our learned refinement policies and graph search heuristics. For the dinner domain, we report results only for the first 2 systems, because the errors propagated in this domain relate to the stackability of objects. Since this is independent of continuous parameter instantiations, we want to incorporate all available error information when attempting refinement. Thus, depth-first search can be expected to perform well in this setting.

For the third system, which trains a refinement policy and graph search heuristics, we employ the following algorithm to produce a trained set of weights. We train 3 sets independently, test each one on a validation set of 50 environments, and output the best-performing one. We found that this reduced variation due to random seeding. For the second system, by contrast, we train a single set of weights and output it, without any validation.

We report results on a fixed test set of 50 randomly generated environments. For the second and third systems, we average across running the training process 10 times and evaluating each final set of weights separately.

Our experiments are conducted in Python 2.7 using the OpenRave simulator [10] with a PR2 robot. The motion planner we use is trajopt [11], and the task planner is FastForward [12]. The experiments were carried out in series on an Intel Core i7-4790K machine with 16GB RAM. Table I summarizes our quantitative results, and Figure 1 and Figure 2 show learned refinement policies.

B. Can Domain

We run two sets of experiments, using 25 objects and 30 objects on the table. The goal across all experiments is for the robot to pick up a particular object with its left gripper. We disabled the right gripper, so any obstructions to the target object must be picked up and placed elsewhere on the table. This domain has 4 types of continuous parameters: base poses, object grasp poses, object putdown poses, and object putdown locations onto the table.

Our curriculum learning system first trains base poses and grasp poses for $N = 12$ iterations with $\epsilon = 5$, then base poses, grasp poses, and putdown poses (at fixed location) for $N = 18$ iterations with $\epsilon = 20$, then all parameter types for $N = 30$ iterations with $\epsilon = 20$. We fixed $L = 100$.

The results demonstrate significant improvements in performance to the baseline system for success rate, motion planning time, and number of motion planner calls. For the 30 object case, though, there is an increase in average motion planning time when we use trained graph search heuristics. This is likely because of suboptimal search decisions occasionally being made.

C. Dinner Domain

We run two sets of experiments, using 2 and 4 bowls. The robot must move the bowls from their initial locations on one table to target locations on the other. We assign a cost to base motion in the environment, so the robot is encouraged to use the provided tray, onto which bowls can be stacked. This domain has 5 types of continuous parameters: base poses, object grasp poses, object putdown poses, tray pickup poses, and tray putdown poses.

Our curriculum learning system first trains base poses and tray pickup and putdown poses for $N = 20$ iterations, then object grasp and putdown poses for $N = 20$ iterations. We fixed $L = 100$ and $\epsilon = 10$.

The results demonstrate comparable performance to the baseline system. The reason is that hand-coded discretizations of the sample space are very good in this domain. For example, the optimal robot base pose from which to pick up the tray is directly in front of it, which is quickly sampled through the baseline system's discretization.

VIII. Conclusion

We presented a novel application of reinforcement learning to task and motion planning. We gave a randomized local search algorithm for plan refinement and trained policies for it. We also showed initial work that trains heuristics for searching the plan refinement graph, so our full system learns both *which* node to refine and *how* to perform this refinement. We evaluated performance against a baseline of hand-coded discretizations for several challenging tasks; our system demonstrated significantly improved performance.

References

[1] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel, "Combined task and motion planning through an extensible planner-independent interface layer," *IEEE Conference on Robotics and Automation*, 2014.

[2] L. P. Kaelbling and T. Lozano-Pérez, "Hierarchical task and motion planning in the now," in *IEEE Conference on Robotics and Automation*, 2014.

[3] F. Lagriffoul, D. Dimitrov, J. Bidot, A. Saffiotti, and L. Karlsson, "Efficiently combining task and motion planning using geometric constraints," 2014.

[4] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "FFRob: An efficient heuristic for task and motion planning," in *International Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2014. [Online]. Available: http://lis.csail.mit.edu/pubs/garrett-wafr14.pdf

[5] C. Dornhege, P. Eyerich, T. Keller, S. Trüg, M. Brenner, and B. Nebel, "Semantic attachments for domain-independent planning systems," in *Towards Service Robots for Everyday Environments*. Springer, 2012, pp. 99–115.

[6] W. Zhang and T. G. Dietterich, "A reinforcement learning approach to job-shop scheduling," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, ser. IJCAI'95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 1114–1120. [Online]. Available: http://dl.acm.org/citation.cfm?id=1643031.1643044

[7] M. Zucker, J. Kuffner, and J. A. D. Bagnell, "Adaptive workspace biasing for sampling based planners," in *Proc. IEEE Int'l Conf. on Robotics and Automation*, May 2008.

[8] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

[9] S. Chib and E. Greenberg, "Understanding the metropolis-hastings algorithm," *The american statistician*, vol. 49, no. 4, pp. 327–335, 1995.

[10] R. Diankov and J. Kuffner, "Openrave: A planning architecture for autonomous robotics," Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-08-34, July 2008.

[11] J. Schulman, J. Ho, A. Lee, I. Awwal, H. Bradlow, and P. Abbeel, "Finding locally optimal, collision-free trajectories with sequential convex optimization." in *Robotics: Science and Systems*, vol. 9, no. 1. Citeseer, 2013, pp. 1–10.

[12] Jörg Hoffman, "FF: The fast-forward planning system," *AI Magazine*, vol. 22, pp. 57–62, 2001.