Learning Proposal Distributions for Refining Symbolic Plans

Rohan Chitnis[1], Dylan Hadfield-Menell[2], and Pieter Abbeel[2]

***Abstract*— A challenge in mobile manipulation planning is the length of the horizon that must be considered; 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. A core limitation of these systems is the manner in which continuous parameters for motion planning are sampled: using hand-coded distributions that leverage domain specificity, lack robustness, and require substantial design effort. In this paper, we present a method for using reinforcement learning (RL) to learn distributions that propose values for the continuous parameters of a symbolic plan. More specifically, we formulate *plan refinement*, the process of determining continuous parameter settings for a fixed task sequence, as a Markov decision process (MDP) and give an algorithm to learn a policy for it. Additionally, we show how to train heuristics that guide meta-level search through a *plan refinement graph*, a data structure whose nodes store candidate symbolic plans for reaching the goal state and their current refinements. This allows our system to determine *which* symbolic plans to focus refinement on, in addition to *how* to perform this refinement. Our contributions are as follows: 1) we present a randomized local search algorithm for plan refinement that is easily formulable 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 searching through a plan refinement graph; and 4) we perform experiments to evaluate the performance of our system in a variety of simulated domains. We find that our approach yields significantly improved performance over that of hand-coded sampling distributions.**

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 (e.g., setting a dinner table, doing laundry) over long horizons. 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. The hierarchical separation of high-level, symbolic task planning from low-level, geometric motion planning enforces abstraction: the task planner maintains no knowledge of the environment geometry, and the motion planner has no understanding of the overall goal. Efficient integration of these two types of reasoning is challenging, and recent research has proposed several methods for it [1], [2], [3], [4], [5].

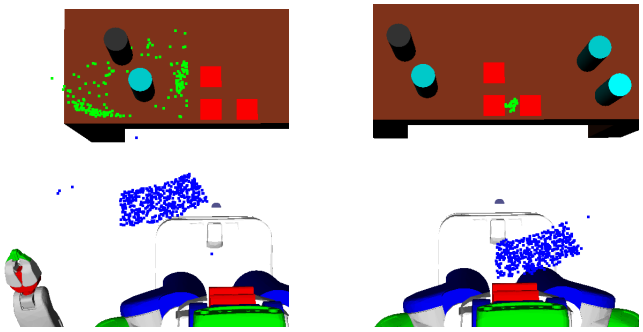

Fig. 1: Screenshots showing some distributions learned by our system in a simulated pick-and-place domain. We use reinforcement learning to train good sampling distributions for continuous motion planning parameters in long-horizon tasks. The robot is tasked with grasping the black cylinder and putting it down on the central red square. The left image shows learned base motion (blue) and grasping (green) distributions, and the right shows learned base motion (blue) and putdown (green) distributions. The grasping policy learned automatically to avoid the region close to the light blue obstruction. These distributions are optimized to reduce the number of motion planner calls needed to produce a complete plan.

A key limitation of TAMP systems is that they typically rely on hand-coded discretization when sampling continuous parameter values, such as target grasp poses, for motion planning. Designing these heuristic sampling distributions often requires substantial effort, as it necessitates a parametrization of the trajectories a robot must follow to perform actions in its environment. Often, these parametrizations are tuned based on geometric attributes of the environment and its objects, meaning that they must be recalibrated when running the system in a new setting. Further, such discretizations must be fairly coarse to allow reasonable search speeds, meaning they inherently lack robustness to increased environmental complexity. In this paper, we present a reinforcement learning method to train good proposal distributions for sampling continuous parameter values intelligently. The distributions we learn are not discrete, and they implicitly parametrize feasible robot trajectories. We also develop a method for making the meta-level decision of which symbolic task sequence to sample for, given options that stem from several potential logical states of the world.

[1] ronuchit@berkeley.edu
[2] {dhm, pabbeel}@eecs.berkeley.edu

Our methods build on the TAMP system presented by Srivastava et al. [1]. In this system, 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, thus grounding it. These candidate values are checked locally for feasibility by calling a motion planner.

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 hand-coded discrete set of candidate parameter values. 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. This plan is added as a child node in the *plan refinement graph*, whose nodes store each symbolic plan and associated current refinement. This graph's search policy determines the order in which plans are attempted to be refined. The system uses an off-the-shelf classical task planner and motion planner, both as black boxes. While our method is specific to this architecture, it can be adapted for other TAMP paradigms.

Reinforcement learning (RL) refers to the process of an 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 apply RL to 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.

We train a policy that determines how to sample values for plan refinement, using policy optimization with linear function approximation. Our approach allows the learned distributions to be continuous, robust to changes in the environment, and trainable for any experimental setting, eliminating the need for hand-tuned distribution parameters. Also, we train heuristics that encode a search policy through a plan refinement graph, thus providing an ordering for 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 formulable 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 discretization of the plan refinement sample space.

II. Related Work

Our work uses RL techniques to improve upon results in recent TAMP systems. We train good policies for sampling plan variable values, whereas other systems use hand-coded heuristics to guide this sampling.

Srivastava et al. [1] hand-tune coarse, domain-specific discretizations for determining the robot pose in actions such as moving the base and picking up an object from a table. Kaelbling et al. [2] use "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. 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. 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 to these systems, our method learns continuous sampling distributions for plan variables, which are significantly more robust to different environment layouts.

Prior work has been devoted to using machine learning techniques for training heuristics to guide search algorithms. This general formulation has been applied to many domains other than hierarchical planning for robotics.

Boyan et al. [8] present an application to solving optimization problems using local search routines. They describe an algorithm, STAGE, for learning a state evaluation function using features of the optimization problem. This function then guides the local search toward better optima.

Arbelaez et al. [9] use machine learning to solve constraint satisfaction problems (CSPs). Their approach, Continuous Search, maintains a heuristic model for solving CSPs and alternates between two modes. In the *exploitation* mode, the current heuristic model is used to solve user-inputted CSPs. In the *exploration* mode, this model is refined using supervised learning with a linear SVM.

Xu et al. [10] apply machine learning for classical task planning, drawing inspiration from recent advances in discriminative learning for structured output classification. Their system trains heuristics for controlling forward state-space beam search in task planners.

To our knowledge, our work is the first to use RL for plan refinement in a TAMP system.

III. Background

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

A. Reinforcement Learning

A reinforcement learning (RL) problem is typically formulated as a Markov decision process (MDP). An MDP is defined by the following:

- A state space $\mathcal{S}$.
- An action space $\mathcal{A}$.
- A transition function $P(s'|s,a)$ for $s, s' \in \mathcal{S}, a \in \mathcal{A}$.

- A reward function $R(s, a, s')$ for $s, s' \in \mathcal{S}, a \in \mathcal{A}$.
- A discount factor $\gamma \in [0, 1]$.

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. The goal of an RL problem is for an agent to learn π^* (or an approximation) automatically through interaction with its environment: taking actions within it and obtaining rewards. Key challenges in RL include determining how to assign the credit of received rewards to each undertaken action, balancing the tradeoff between exploration of new actions and exploitation of known good actions in the environment, and handling potentially limited perception of the current state. Widely used techniques for RL include value function approximation, using methods such as temporal difference (TD) learning [11], and direct policy estimation, which encompasses both gradient-based and gradient-free methods.

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 search as an MDP as follows:

- $\mathcal{S}$ is the set of all possible temporal configurations and resource allocations of the jobs.
- $\mathcal{A}$ is the set of all possible changes to either the time or the resource allocation of a single job.
- R is a measure of schedule goodness based on its length relative to the difficulty of the scheduling problem.

They use $TD(\lambda)$ [11] with function approximation to learn a value function for this MDP, allowing infeasible schedules. Their approach outperforms the previous state of the art for this task; it also scales better to larger scheduling problems after having been trained only on smaller ones. This application of RL to planning has been built on extensively in the literature, applied to domains such as retailer inventory management [12], single-agent manufacturing systems [13], and multi-robot task allocation [14].

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. They use a variant of the REINFORCE family of stochastic policy gradient algorithms to learn a good weight vector for these features. Their experimental results, which focus on biasing sampling distributions for the bidirectional RRT algorithm, demonstrate reduced motion planning time for a variety of scenarios involving path planning through obstructed environments. In our work, we adopt their policy gradient updates for the TAMP framework (Section V-C).

C. Task and Motion Planning

A fully observed task planning problem is a tuple $(\mathcal{O}, \mathcal{T}, \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{C})$:

- $\mathcal{O}$ is a set of symbolic references to *objects* in the environment, such as cylinders, cylinder locations, 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. All others are assumed false.
- $\mathcal{G}$ 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 fluents characterizing what must hold true in the current state for the action to be validly performed, and *effects*, a set of fluents 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 to the state of fluents successively starting with $\mathcal{I}$, results in a state in which all fluents of $\mathcal{G}$ hold true. We refer to this sequence of actions as a *symbolic*, or *high level*, plan.

Since task planners operate on a symbolic level, they are unable to maintain a geometric interpretation for their continuous variables, such as grasping poses for objects. Thus, we use TAMP to provide this interpretation. We adopt the hierarchical method of Srivastava et al. [1], a modular approach to TAMP that uses a black-box task planner and motion planner. 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.

An *interface layer* is then responsible for assigning continuous values to instantiate these symbolic references and calling a motion planner to find valid linking trajectories for these instantiated values. In the process, the interface layer may discover geometric facts about the environment which cannot be resolved by motion planning (such as finding that an object to be grasped is obstructed), upon which it sends symbolic representations of these facts back to the task planner to be added into the fluent state. *Plan refinement* refers to this process of attempting to assign a feasible set of values to the symbolic plan variables. In this context, feasibility means that there exists a set of collision-free trajectories for the robot that links all instantiated values associated with robot pose.

The system also maintains a *plan refinement graph*, whose nodes each store a symbolic plan and its current refinement, the set of instantiated plan variable 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. Thus, each node in the graph corresponds to a plan constructed from a fluent state incorporating some distinct subset of all facts discovered by the interface layer so far. The search policy through the plan refinement graph enforces an ordering over which refinement is attempted for plans. In the existing infrastructure, this policy always selects the deepest node for refinement, meaning all propagated errors are incorporated into the plan. (Work on this portion of the system is still in progress and will be published in the future.)

The authors introduce an algorithm for plan refinement called TRYREFINE, an exhaustive backtracking routine over a hand-coded 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 domain, end effector grasp poses for a can are sampled around the can in each of the 4 cardinal 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.

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], by initializing an infeasible refinement and continually making local improvements. We thus present one contribution of our work: randomized refinement, a local search algorithm for plan refinement. It maintains at all times 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 (locally). Algorithm 1 shows pseudocode for 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. We continue sampling until we find bindings for symbolic pose references that satisfy inverse kinematics constraints (IK feasibility). Trajectories are then initialized as straight lines.

We call the MOTIONPLAN subroutine in line 4, which iterates through the sequence of actions that comprise the high level plan (l.20). For each action, a call to the motion planner is made (l.22) using the instantiated values for the parameters of that action, to attempt to find a trajectory linking the sampled poses. If this succeeds, the preconditions of the action are tested (l.26); as part of this step, we check if the trajectory returned by the motion planner is collision-free.

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.

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:              return success
7:          end if
8:          if failPred is null then
9:              # Motion planning failure.
10:             failAction ← HLP.ops[failStep]
11:             resample(failAction.params)
12:         else
13:             # Action precondition violation.
14:             resample(failPred.params)
15:         end if
16:     end for
17:     Raise failure to task planner, receive new plan.
18: end procedure
19: procedure MOTIONPLAN(HLP)
20:     for op_i in HLP.ops do
21:         res ← op_i.motionPlan()
22:         if res.failed then
23:             return i, null
24:         end if
25:         failPred ← op_i.checkPreconds()
26:         if failPred is not null then
27:             return i, failPred
28:         end if
29:     end for
30:     # Found successful plan refinement.
31:     return null, null
32: end procedure
```

Randomized refinement has two key properties. The first is a very explicit algorithm state. We show in the next section that this allows for a straightforward MDP formulation. This is also beneficial from an engineering perspective, as the simplicity allows for easy debugging. The second is that it allows the parameter instantiations for a particular action in the plan to be influenced by those for a *future* action. For example, in a pick-and-place task, it can make sense for the object's grasp pose to be sampled conditionally on the current instantiation of the putdown pose, even though the putdown appears after the grasp in the plan sequence. Thus, it is easy for plan refinement to respond to long-term dependencies in continuous values of plan variables.

V. LEARNING REFINEMENT DISTRIBUTIONS

In this section, we present our primary contribution: an RL approach that learns a policy for plan refinement. Before describing our algorithm, we show how to formulate the problem as an MDP.

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), consisting of the symbolic plan, its current refinement (instantiation of values for all plan parameters), and the environment.
- An action $a \in \mathcal{A}$ is a pair (p, x), where p is the discrete-space plan parameter to resample and x is the continuous-space value assigned to p in the new refinement.
- The transition function is defined implicitly by the motion planning algorithms we use as a black box.
- The reward function $R(s, a, s')$ provides rewards based on closeness to a valid plan refinement.
- $\mathcal{P}$, a distribution over planning problems, encompassing both the task planning problem and the environment. This defines the initial state distribution for the MDP.
- L, the number of samples for one planning problem, which partially defines the transition function for the MDP. (After L calls to the sampler, the next state is drawn uniformly at random from $\mathcal{P}$.)

We are focused on training policies for choosing the continuous value x in actions $a \in \mathcal{A}$, by defining a sampling distribution (based on the state) for each plan parameter p. We note that randomized refinement provides a fixed policy for selecting p itself.

Our reward function R explicitly encourages successful plan refinement, providing positive reward linearly interpolated between 0 and 20 based on the fraction of the plan variables whose current instantiation causes action preconditions to be met. 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 learns a mapping from a feature vector to a likelihood. In our setting, we learn a weight vector θ_p for each parameter *type*, comprised of a pose type and a gripper (e.g., "left gripper grasp pose," "right gripper putdown pose"). This decouples the learned distributions from any single high level plan and allows us to test in more complicated environments than those used in training. These weight vectors encode the policy for selecting x in actions $a \in \mathcal{A}$.

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. As the system takes on-policy actions $a \in \mathcal{A}$ (based on current θ), collect rewards according to R. After every ϵ calls to `resample`, perform a gradient update on the weights using the collected rewards.

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 [15]. 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. Searching the Plan Refinement Graph

The approach presented thus far can be succinctly described as learning *how* to refine a single high level plan. We are also interested in learning heuristics for the meta-level decision of searching through the plan refinement graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, which grows as geometric facts about the environment are discovered by the interface layer and propagated to the task planner to be added into the fluent state (Section III-C). Each node $v \in \mathcal{V}$ contains a high level plan and its current refinement. 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 instead try to discover geometric facts to incorporate into the fluent state. As an example, action 2 is useful if the plan attempts to grasp an object that is surrounded by obstructions, so that no valid refinement of the grasping pose exists.

We train two decision tree regressors which approximate answers for 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 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'? Because it is challenging to predict answers to these questions directly, we estimate by instead training regressors that answer these two questions for a manipulation action involving a *single* object in the environment, then approximating the desired quantities for an entire high level plan by summing the predicted values

# Objects	System	% Solved (SD)	Avg MP Time (s)	Avg # MP Calls
2 (dinner)	B	TODO (X)	0	0
2 (dinner)	L	TODO (X)	0	0
4 (dinner)	B	TODO (X)	0	0
4 (dinner)	L	TODO (X)	0	0
25 (cans)	B	74 (0)	15.8	19.7
25 (cans)	L	82 (4.8)	15.6	14.6
25 (cans)	F	92 (4.2)	11.2	12.4
30 (cans)	B	36 (0)	48.8	35.8
30 (cans)	L	65 (6.1)	24.6	9.7
30 (cans)	F	74 (5.5)	23.3	21.7

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 system with only learned refinement policies (no graph search heuristics) (L), and our full system (F). Results using our system are averaged across 10 different sets of weights. Time limit: 300s.

over all manipulations that occur throughout. This approximation depends heavily on the localized nature of plans in our domain; it only makes sense if a plan can be split into subportions whose refinements are mostly independent. Addressing this will be an important area of future work.

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. To train the regressors, we fix pretrained policies for plan refinement and construct supervised learning datasets 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 quantities of interest. For the second regressor, we do the same but on a single child node spawned from the root node. We then fit a standard decision tree regressor to our data.

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 – if producing a child node would reduce the number of steps to a valid refinement, we bias toward selecting action 2 for v. Note that this procedure treats $\mathcal{V}$ as an unordered set and ignores the structure of $\mathcal{G}$.

VII. Experiments

Our experiments are conducted in Python 2.7 using the OpenRave simulator [16] with a PR2 robot. The motion planner we use is trajopt [17], and the task planner is FastForward [18]. The experiments were carried out in series on an Intel Core i7-4790K machine with 16GB RAM.

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 that of the hand-coded sampling distributions and default plan refinement graph search strategy used in Srivastava et al. [1]. In both domains, plan refinement samples are drawn from continuous distributions trained using the described algorithms. For the can domain, we show results both with and without our trained heuristics for searching the plan refinement graph. However, for the dinner domain, objects in the environment are never obstructed by others, so the default search policy of picking the deepest node in the graph to refine is already optimal.

We use the reward function described earlier. Our weight vectors are initialized to $\vec{\mathbf{0}}$ for all parameter types – this initialization represents a uniform distribution across the limits of the geometric search space. We use 24 features. 9 binary features encode the bucketed distance between the sample and target. 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 $3\pi/4$.

For the can domain, 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. The range of allowable values for grasp and putdown poses is a cube of side length 30 centimeters around the target object or putdown location. For base poses, the range is a square of side length 1 meter around the object or location which the robot is approaching. For putdown locations, the range is defined by the edges of the table.

Initial experimentation revealed that training jointly for all these continuous parameter types was infeasible due to a dearth of valid plan refinements being produced, so we decided to incorporate curriculum learning. Thus, we train $N = 10$ iterations of plans involving just base motion and grasping, then $N = 20$ iterations of base motion, grasping, and putdown at a predefined location, then $N = 30$ iterations of the full set of parameter types, including a varying putdown location. We fixed $L = 100$ and $\epsilon = 20$.

For the experiments which only use the trained plan refinement policies (no learned heuristics for searching the plan refinement graph), we trained 10 different sets of weights using the described curriculum learning system, varying only the random number generator's initial seed. Then, for the experiments which also search the plan refinement graph intelligently, we employed an improved training methodology to reduce variation due to seeding. We trained 30 different sets of weights, and for every group of 3 sets, we picked the best one based on performance in a validation set and

discarded the other two. To test performance, we randomly generated 50 environments for both 25 cans and 30 cans on the table, and measured success rate, motion planning time, and number of motion planner calls for the baseline system and each trained set of weights.

For the dinner domain, we run two sets of experiments, in which the robot must move 2 and 4 bowls from their initial location 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 the tray pickup and putdown poses for $N = 20$ iterations, then the object grasp and putdown poses for $N = 20$ iterations. We fixed $L = 100$ and $\epsilon = 10$. We trained 10 different sets of weights (varying only the seed) and tested performance across 50 randomly generated environments for each number of objects.

Table I summarizes our quantitative results, which demonstrate comparable performance to the baseline system for the dinner domain, and significant improvements for the can domain. The reason performance does not greatly improve for the dinner domain is that the hand-coded discretizations of the sample space are very good in this environment. For example, the optimal robot base pose from which to pick up the tray is directly in front of it, which is quickly found through the baseline system's discretization for the tray pickup pose parameter.

VIII. Conclusion

We presented a novel application of reinforcement learning to plan refinement in task and motion planning. Our method trained a policy for refining symbolic plans by learning good sampling distributions for plan parameters. The choice of which parameter to resample at each iteration was governed by randomized refinement, a local search algorithm we presented for plan refinement. We additionally trained heuristics for search through the plan refinement graph, so our full system learns both *which* node to refine and *how* to perform this refinement. We evaluated performance by comparing against a baseline of hand-coded discretizations for several challenging tasks, in which 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] J. Boyan and A. W. Moore, "Learning evaluation functions to improve optimization by local search," *J. Mach. Learn. Res.*, vol. 1, pp. 77–112, Sep. 2001. [Online]. Available: http://dx.doi.org/10.1162/15324430152733124

[9] A. Arbelaez, Y. Hamadi, and M. Sebag, "Continuous search in constraint programming," in *Autonomous Search*, Y. Hamadi, E. Monfroy, and F. Saubion, Eds. Springer Berlin Heidelberg, 2012, pp. 219–243. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21434-9_9

[10] Y. Xu, S. Yoon, and A. Fern, "Discriminative learning of beam-search heuristics for planning," in *Proceedings of the International Joint Conference on Artificial Intelligence*, 2007, pp. 2041–2046.

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

[12] B. Van Roy, D. P. Bertsekas, Y. Lee, and J. N. Tsitsiklis, "A neuro-dynamic programming approach to retailer inventory management," in *Decision and Control, 1997., Proceedings of the 36th IEEE Conference on*, vol. 4. IEEE, 1997, pp. 4052–4057.

[13] Y.-C. Wang and J. M. Usher, "Application of reinforcement learning for agent-based production scheduling," *Engineering Applications of Artificial Intelligence*, vol. 18, no. 1, pp. 73 – 82, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0952197604001034

[14] T. S. Dahl, M. Matarić, and G. S. Sukhatme, "Multi-robot task allocation through vacancy chain scheduling," *Robotics and Autonomous Systems*, vol. 57, no. 6, pp. 674–687, 2009.

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

[16] 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.

[17] 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.

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