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 and require substantial design effort. In this paper, we present a method that uses 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 this problem. Our contributions are as follows: 1) we present a randomized local search algorithm for plan refinement that is well-suited to an MDP formulation; 2) we give an RL algorithm that learns policies for this MDP; and 3) we perform experiments to evaluate the performance of our system in a simulated pick-and-place domain. We find that our approach achieves comparable to improved performance versus that of hand-coded sampling distributions in a variety of test environments.**

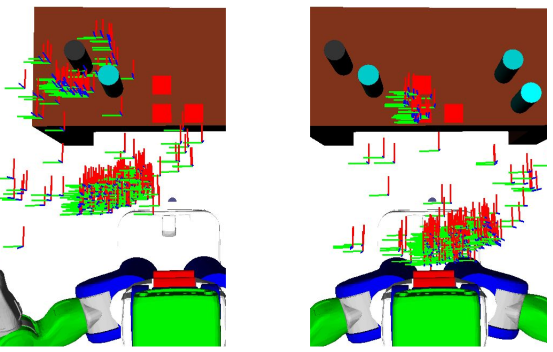


Fig. 1: Screenshots showing some distributions learned by our system in a simulated pick-place domain. We use reinforcement learning to train good distributions for sampling continuous parameters for motion planning. The robot is tasked with grasping the black cylinder and putting it down on the central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

I. Introduction

A long-term goal of robotics research is the introduction of intelligent household robots. To be effective, such a robot 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 apparent the need for a hierarchical system of reasoning.

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].

A key limitation of TAMP systems is that they typically rely on hand-coded discretization to sample continuous parameter values, such as target grasp poses, for motion planning. These heuristic sampling distributions are tuned to the specific geometric properties of the environment and its objects, and designing them well requires substantial effort. This has several negative implications: the parameters of the hand-coded distributions must be fine-tuned when running the system in a new setting, and the resulting refinement distributions lack robustness. In this paper, we present a reinforcement learning method to train good proposal distributions for sampling continuous parameter values intelligently.

Our solution builds on the TAMP system presented by Srivastava et al. [1]. 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 then 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 plan variable values. If a motion planning feasible

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

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 system uses an off-the-shelf classical task planner and motion planner, both as black boxes. We build upon this framework because it is easy to separate the plan refinement module from the task planning module. 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 to maximize 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 symbolic plan parameters, 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.

The three contributions of our work are as follows:

1) We present randomized refinement, a local search algorithm for plan refinement. Randomized refinement maintains at all times a set of values for all symbolic plan variables, then at each iteration randomly resamples one whose current instantiation is causing a failure. This naturally guides plan variable resampling intelligently and easily lends itself to RL.
2) We describe how to formulate plan refinement (using randomized refinement) in the RL framework, so that sampling distributions for plan variables can be learned instead of hand-coded. Our formulation optimizes for minimizing the number of calls to the motion planner.
3) We present experiments to evaluate our approach in a variety of test environments involving complicated grasp and putdown actions with obstructions and base motion.

Our experimental results demonstrate that our approach is competitive with hand-coded discretization of the plan refinement sample space, with respect to both motion planning time and number of calls to the motion planner.

II. Related Work

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.

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. They draw 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 $\pi : \mathcal{S} \to \mathcal{A}$ 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 the optimal policy (or an approximation) through interaction with its environment.

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 controlling such an algorithm 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. 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 arm path planning through obstructed environments. In our work, we adopt their policy gradient updates for the TAMP framework.

C. Task and Motion Planning

We formulate a TAMP problem as follows. A fully observed task planning problem is a tuple $(\mathcal{O}, \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{C})$:

$\mathcal{O}$ is a set of symbolic references to *objects* in the environment, such as cylinders, cylinder locations, and grasping poses.

$\mathcal{F}$ 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 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, discovering geometric facts about the environment, and sending 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 continuous values to the symbolic plan variables. The authors introduce an algorithm for plan refinement called TRYREFINE, an exhaustive backtracking routine over a hand-coded discrete set of feasible plan parameter values. (Interested readers are referred to [1] for details.) These hand-coded discretizations are domain-specific and tuned to the geometry of objects in the environment; for example, in a pick-place domain, end effector grasp poses for a can may be sampled around the can in each of the 4 cardinal directions, at a fixed distance and height. Such sampling distributions must be tuned for each object and domain separately and are not robust to increased environmental complexity.

IV. RANDOMIZED REFINEMENT

Before we can apply RL to plan refinement, we need to 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. This refinement strategy is straightforward to formulate in the RL framework, as we show in the next section. 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. 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 verify that the trajectory returned by the black box motion planner is collision-free, satisfying the precondition that the trajectory is feasible in the environment.

We call the `resample` routine on the high level parameters associated with the 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 classical planner, which will update its fluent state and provide a new high level plan.

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. We also found that this was 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

Algorithm 1 Randomized refinement.

```
procedure RANDREF($HLP, N$)
    $init \leftarrow$ initRefinement($HLP$)
    for iter = 0, 1, ..., $N$ do
        $failStep, failPred \leftarrow$MOTIONPLAN($HLP$)
        if $failStep$ is null then
            return success
        else
            if $failPred$ is null then
                # Motion planning failure.
                $failAction \leftarrow HLP.ops[failStep]$
                resample($failAction.params$)
            else
                # Action precondition violation.
                resample($failPred.params$)
            end if
        end if
    end for
    Raise failure to task planner, receive new plan.
end procedure
procedure MOTIONPLAN($HLP$)
    for $op_i$ in $HLP.ops$ do
        $res \leftarrow op_i$.motionPlan()
        if $res.failed$ then
            return $i$, null
        else
            $failPred \leftarrow op_i$.checkPreconds()
            if $failPred$ then
                return $i$, $failPred$
            else
                # Found successful plan refinement.
                return null, null
            end if
        end if
    end for
end procedure
```

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

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 to assign to p.
- The transition function is implicitly defined by $\mathcal{S}$ and $\mathcal{A}$.
- Our reward function $R(s, a, s')$ provides constant rewards based on parameter resampling.

We are focused on training policies for choosing the continuous value x in our actions, 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.

Because we check for IK feasibility when resampling, we provide -1 reward every time we sample an IK-infeasible pose, and $+3$ reward for an IK-feasible pose. Similarly, we give -3 reward upon motion planning failure or action precondition violation, and $+5$ reward upon a successful call to the motion planner for an action. This choice of R implies that our training optimizes the sampling distributions for motion planning feasibility, rather than explicitly for refining plans. However, the two are closely correlated – randomized refinement succeeds in fewer iterations when the sampling distributions are biased toward motion planning feasible values.

B. Training Process

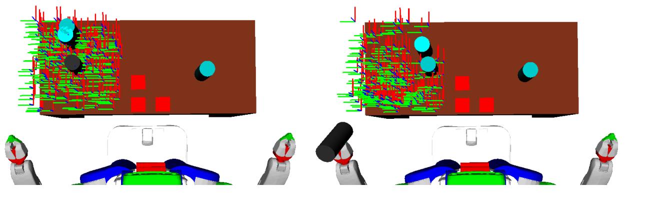
We now show how to apply the method of Zucker et al. [7] to learn a policy for plan refinement. Their method learns a weight vector that maps a feature vector to a likelihood. In our setting, we learn a weight vector for each parameter *type*, comprised of a pose type and a gripper (e.g., “left gripper 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 we trained. The weight vectors encode the policy for selecting x for actions $a \in \mathcal{A}$, as described in the previous subsection. Our feature function $f(s, p, x)$ 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 our problem.

Our training system is defined by the following:

$\mathcal{P}$:	A distribution over planning problems, encompassing both the task planning problem and the environment.
θ_p:	The weight vector for p's parameter type.
R:	The reward function, described in the previous subsection.
$f(s, p, x)$:	Described above.
N:	The number of planning problems on which to train.
L:	The number of samples for one planning problem.
ϵ:	The number of samples comprising an episode. We perform a weight vector update after each 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. As the system takes on-policy actions $a \in \mathcal{A}$ (based on current θ values), collected rewards are stored. Every ϵ calls to `resample`, we perform a gradient update on the weight vectors using the collected rewards.

This process shows the strength of randomized refinement: only the variable values responsible for failures are resampled, so the learning is naturally focused toward improving


(a) Initial distribution (b) After 5 iterations.

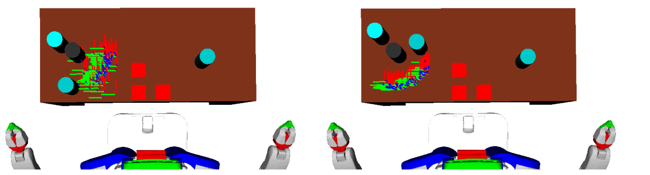

(c) After 15 iterations. (d) Final distribution.

Fig. 2: Learned left arm grasping distributions used to pick up the black can, after different training iterations. An iteration refers to a single run of randomized refinement, which terminates after L calls to the resample function. The initial distribution is uniform because we initialize weights to $\vec{\mathbf{0}}$. The final distribution is after 20 iterations. The convergence of the distribution demonstrates the soundness of our approach.

sampling distributions which are not yet well-shaped.

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

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

They then define the expected reward of an episode ξ:

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

and provide an approximation for its gradient:

$$\nabla\eta(\theta_p) \approx \frac{R(\xi)}{\epsilon} \sum_{i=0}^{\epsilon} (f(s, p, x_i) - \mathbb{E}_{q,s}[f]).$$

$R(\xi)$ 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. Experiments

We evaluate our approach in several pick-and-place tasks, varying the number of obstructions, obstruction locations, and whether the base is allowed to move. We compare performance with that of the hand-coded sampling distributions used in Srivastava et al. [1]. In this work, our training focuses on the problem of refining a single high level plan; however, during actual planning, we may start with a high level plan that has no valid refinement, and symbolic error information must be propagated to the task planner, which produces a new plan. In future work, we will incorporate this into the learning framework, but for now we measure performance only on refining the final high level plan, which has a valid refinement.

We use the reward function described earlier. Our weight vectors are initialized to $\vec{\mathbf{0}}$ for all pose parameter types. The initialization represents a uniform distribution across the limits of the geometric search space. Our domain has 3 types of continuous parameters: base poses, object grasp poses, and object putdown poses. 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. 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

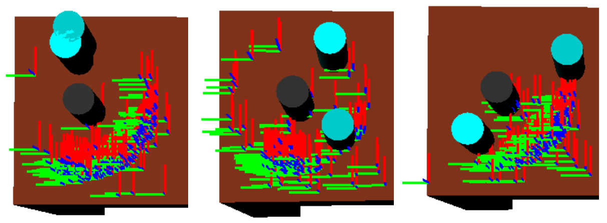

(a) (b) (c)

Fig. 3: Our learned grasp pose distribution is shown with different obstruction layouts. The system learns to avoid obstructions while providing a good set of grasping poses.

System	Experiment	% Solved	Avg MP Time (s)	Avg # MP Calls
B	1, 1 obstr	100	8.16	18.93
L	1, 1 obstr	96	6.11	11.5
B	1, 2 obstr	96	8.74	19.7
L	1, 2 obstr	94	12.26	20.77
B	1, 3 obstr	84.44	10.77	23.16
L	1, 3 obstr	97.78	10.6	19.1
B	2	33.33	12.1	21.1
L	2	96.67	9.19	17.78
B	3	100	4.52	12.24
L	3	100	5.94	15.36

TABLE I: Percent solved, along with time spent motion planning and number of calls to the motion planner for the final refinement. Results are averaged across 50 test environments per experiment, when both the baseline and our system succeed. The test environments were grouped into 10 batches of 5, and each batch received independent training with a distinct random seed. For each experiment, we provide baseline system results using hand-tuned sampling distributions (B) and results from running our system (L).

number of other objects within discs of radius 7, 10, and 15 centimeters around the target. 3 binary features encode 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 $3\pi/4$.

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 Fast-Forward [18]. The experiments were carried out in series on an Intel Core i7-4770K machine with 16GB RAM.

We train and test 3 different experiments, all of which share a common goal of grasping a target object from a table and putting it down at a specific location on the table. The locations of the obstructions surrounding the target object were sampled uniformly from a thick ring around the object with inner radius 0.13 meters and outer radius 0.25 meters. Table I summarizes our quantitative results. Figure 1 plots learned distributions for a move-grasp-move-putdown sequence of actions. Figure 2 illustrates the training of a grasp distribution, and Figure 3 shows how the learned distribution interacts with obstructions.

Experiment 1: Grasp is obstructed by 1 through 3 objects; no base motion. $N = 20$, $L = 16$, $\epsilon = 4$. Our system outperforms the baseline for both 1 and 3 obstructions, but performs worse for 2. We attribute this to the fact that random seeding can have a huge effect on which local optimum our policy gradient algorithm finds. An additional observation is that motion planner time and number of calls are not always linearly related; some target poses may be IK-feasible but difficult to reach, causing the motion planner to take longer to find a collision-free trajectory.

Experiment 2: Putdown is obstructed in cardinal directions; no base motion. This is a contrived example to illustrate the issues with coarse discretization of the sample space. $N = 20$, $L = 16$, $\epsilon = 4$. The baseline system only attempts putdown pose instantiation in the cardinal directions, explaining the large performance disparity.

Experiment 3: Grasp is obstructed by 1 object; base motion allowed. $N = 60$, $L = 100$, $\epsilon = 20$. Our system performs slightly worse than the baseline here, but not significantly. We believe this is because more time is spent motion planning base poses for which object grasp and putdown are infeasible (since candidate base poses are no longer hand-engineered), triggering resampling of the base pose parameter.

VII. Conclusion and Future Work

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 a novel refinement strategy we presented called randomized refinement. We evaluated performance by comparing against a baseline of hand-coded distributions for several challenging pick-and-place tasks. Our system demonstrated overall comparable to improved performance in motion planning time and number of motion planner calls.

In future work, we plan to improve the feature vector to incorporate information about the symbolic plan and previous samples of a parameter. Additionally, we hope to to extend our RL formulation to learn which parameter to resample, in addition to how to resample it. Finally, we plan to extend our formulation to learn a policy for returning error information to the task planner, instead of relying on reaching the iteration limit in the randomized refinement algorithm.

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 *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE, 2011, pp. 1470–1477.

[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*, 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.