Guided Search for Task and Motion Planning Using Learned Heuristic Functions

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. In this paper, we present machine learning techniques to improve the reliability of planning in TAMP. The system we build on first plans symbolically, and then performs *plan refinement*, which determines feasible settings of values for symbolic references to continuous parameters. Our methods are adaptable for other systems. We present two improvements to current TAMP systems. First, to account for continuous parameters, these systems typically rely on hand-coded distributions to reduce the sample space. Such an approach lacks robustness and requires substantial design effort. We thus use reinforcement learning (RL) to train policies for plan refinement. Second, we develop a complete algorithm for TAMP by introducing a *plan refinement graph*, which allows interleaving plan refinement with a search over *which* plan to try refining, given options that address different infeasibilities. We then use machine learning to train heuristic functions that guide this search. Our contributions are as follows: 1) we present a complete algorithm for TAMP; 2) we present a randomized local search algorithm for plan refinement that is easily formulated as a Markov decision process (MDP); 3) we give an RL algorithm that learns a policy for this MDP; 4) we present a method that trains heuristics for intelligently searching a plan refinement graph; and 5) we perform experiments to evaluate the performance of our system in a variety of simulated domains. We show significant improvements in success rate over the system we build on.

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 difficult, and recent research has proposed several methods for it [1], [2], [3], [4], [5].

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

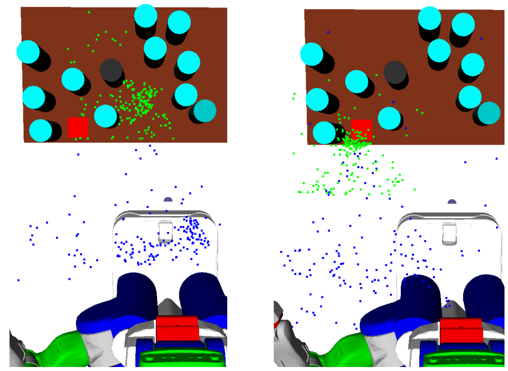


Fig. 1: Screenshots showing distributions learned by our system in a simulated pick-and-place domain. We use reinforcement learning to train good sampling distributions for continuous motion planning parameters in long-horizon tasks. The robot must grasp the black can and put it down on the red square. 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.

We adopt the principles of abstraction in the TAMP system developed by Srivastava et al. [1] (henceforth referred to as SFRCRA-14) to factor the reasoning and search problems into interacting logic-based and geometric components.

SFRCRA-14 uses a (classical) task planner to produce 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. On failure, the logic-based layer of abstraction is enriched with the information gained from geometric reasoning, and the process is repeated.

The system conducts a greedy search to find a symbolic plan for which a motion planning feasible refinement is possible. Although this algorithm achieves good results in a class of domains, it is complete only in some situations; in general, it may not find solutions to solvable problems.

The TAMP paradigms cited above use hand-coded heuristics to instantiate symbols with continuous values. The search for instantiations of symbolic references typically uses domain-specific insight from the user to reduce the space of possible values. For example, in SFRCRA-14, end effector grasp poses for a can in a pick-and-place task are sampled around the can in each of the 8 cardinal and intermediate directions, at a fixed distance and height. Such coarse distributions require fine-tuning for each domain and lack robustness to increased environmental complexity.

In this work, we develop a complete algorithm for TAMP by introducing a *plan refinement graph*, which allows interleaving plan refinement (the search for symbolic reference instantiations) with a search over *which* symbolic plan to try refining, given options that address different infeasibilities. Furthermore, we present machine learning techniques to train heuristic functions that guide both of these search processes.

We use reinforcement learning (RL) to train continuous proposal distributions for plan refinement, thus avoiding the discretization common to TAMP systems. Reinforcement learning 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. We take inspiration from their approach by applying RL to plan refinement. 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.

The five contributions of our work are as follows: 1) we present a complete algorithm for TAMP by maintaining a plan refinement graph; 2) we present randomized refinement, a local search algorithm for plan refinement that is easily formulated as an MDP; 3) we formulate plan refinement in the RL framework and learn a policy for this MDP; 4) we train heuristics to search intelligently through a plan refinement graph; and 5) 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 SFRCRA-14.

II. Related Work

Our work uses machine learning techniques to improve planning reliability 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.

Another line of 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 machine learning to guide search in a TAMP system.

III. Background

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

A. Task and Motion Planning

A motion planning problem is a tuple $\langle \mathcal{X}, f, p_0, p_t \rangle$, where $\mathcal{X}$ is the space of possible configurations or poses of a robot, f is a Boolean function that determines whether or not a pose is in collision and $p_0, p_t \in C$ are the initial and final poses. Without loss of generality, for brevity we assume that $\mathcal{X}$ includes the poses of every movable object. A TAMP problem adds more abstract concepts to this formulation, including *fluents* (logical properties that may vary across configurations) and *actions* (operations that the agent may choose to execute, resulting in changes in the configuration and set of fluents that are true). Each action may require motion planning prior to execution. The overall problem is to plan the sequence of actions that the agent can execute in order to achieve a desired goal condition expressed in terms of fluents.

For example, the action *grasp(Object o, Manipulator p, GraspingPose g, Trajectory m)* indicates the operation of grasping an object o. In order to apply an action, the agent must choose instatiations for the action arguments. An *instantiated* action, such as *grasp(*can_1*, left,* g_1*,* m_1*)*, where the variables have been replaced by names of specific elements of the corresponding type, changes the truth of specific fluent instantiations (also called fluent *literals*), such as *empty(*$gripper_l$*)*.

Definition 1: Formally, we define the task and motion planning (TAMP) problem as a tuple $\langle \mathcal{O}, \mathcal{T}, \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{U} \rangle$:

- $\mathcal{O}$ is a set of objects denoting elements such as cans, bowls, trajectories, and poses. Note that $\mathcal{O}$ includes the

configuration space of all movable objects including the robot.
- $\mathcal{T}$ is the set of object *types*, such as cylinders, motion plans, poses, and locations.
- $\mathcal{F}$ is a set of *fluents*, which define relationships among objects and are Boolean functions defined over the configuration space.
- $\mathcal{I}$ is the set of fluent literals that hold true in the initial state.
- $\mathcal{G}$ is the set of fluent literals defining the goal condition.
- $\mathcal{U}$ is a set of *high-level actions* that are parameterized using objects and 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.

An instantiated action is said to be *feasible* in a state iff its preconditions hold in that state. Actions that include trajectories as arguments have a default precondition that the trajectory must be collision-free.

A solution to a TAMP problem is a sequence of instantiated actions $a_0, a_1, ..., a_n \in \mathcal{U}$ such that every action is feasible when it is applied on states successively starting with $\mathcal{I}$, and the state achieved at the end of the execution sequence satisfies the goal condition $\mathcal{G}$.

B. 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.

Definition 2: Formally, 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, $\pi : \mathcal{S} \to \mathcal{A}$, 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.

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 [11].

C. 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 $TD(\lambda)$ [11] 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 VI-C).

IV. Solving Task and Motion Planning Problems

Solving TAMP problems requires evaluation of possible courses of action comprised of different combinations of instantiated action operators. This is particularly challenging because the set of possible action instantiations (and thus the branching factor of the underlying search problem) is infinite. We give a brief overview of SFRCRA-14, a recent approach to TAMP, and refer the interested reader to the cited paper for further details. Then, we present a complete algorithm for TAMP that we implemented in the framework of SFRCRA-14.

A. Preliminaries

Intuitively, SFRCRA-14 conducts a greedy search by incrementally searching for a symbolic plan that solves the high-level abstraction of the given TAMP problem; determining a prefix of the plan that has a motion planning feasible refinement; updating the high-level abstraction to reflect the reason for infeasibility; and searching for a new plan suffix from the failure step onwards.

In general, including geometric properties in the logic-based formulation leads to an increase in the number of objects representing distinct poses and/or trajectories. For instance, expressing the fact that a trajectory for grasping *can*$_1$ is obstructed by *can*$_3$ from the current pose of the robot would require setting a fluent of the form *obstructs(can*$_3$*, pose*$_{17877}$*, trajectory*$_{3219}$*, can*$_1$*)* to true in the description of the high-level state. In turn, this would require adding *pose*$_{17877}$ and *trajectory*$_{3219}$ into the set of objects if they were not already included. Unfortunately, the size of the abstracted, logic-based state space grows exponentially with the number of objects, and such an approach quickly leads to unsolvable task planning problems.

SFRCRA-14 addresses this challenge by abstracting the continuous action arguments, such as robot grasping poses and trajectories, into a *bounded* 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* conducts plan refinement, searching for instantiations of continuous values to symbolic references while ensuring action feasibility. The authors introduce an algorithm called TRYREFINE, an exhaustive backtracking search over the hand-coded, discretized space of instantiations for each symbolic reference. The resulting process is able to utilize off-the-shelf task and motion planners while carrying out the necessary exchange of information in a scalable manner. However, the algorithm is not complete in all situations: it may not find a solution to a solvable problem.

B. A Complete Algorithm

We introduce a complete algorithm that maintains a *plan refinement graph*, whose nodes each store a valid symbolic plan and its current set of instantiated symbolic reference values. If the 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. The greedy search used in SFRCRA-14 can be viewed as depth-first search through this graph. (KEEP THIS SENTENCE)

TODO Sid: explain algorithm, simplify algorithm

C. Proof of Completeness[TODO Sid]

D. Guided Search Techniques[TODO Sid]

You can use any guided search technique to explore the PRgraph

V. RANDOMIZED REFINEMENT

As noted in the previous section, we can use a variety of algorithms to implement plan refinement. Since we want to apply RL to learn policies for refinement, we seek an algorithm that allows for easy formulation as an MDP. We thus present randomized refinement. Our approach imitates 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 symbolic reference. At each iteration, a reference whose current value is leading to a motion planning failure or action precondition violation is picked randomly and resampled. Algorithm 11 shows pseudocode for randomized refinement.

The procedure takes two arguments, a high-level plan and a maximum iteration count. In line 1, we initialize all references in the high-level plan by sampling from their corresponding distributions, which can be hand-coded or learned, as described in Section VI. We continue sampling until we find bindings for symbolic pose references that

```
Algorithm Complete TAMP
1   for trial in 1 ... do
2       for j in 1 .. trial do
            /* Traverse graph of plans, initially
               with just one plan: ϵ.                */
3           π ← GETNEXTPLANFROMGRAPH()
4           PROCESSREFS(π, j, ChunkSize)
5           REWEIGHPLANSINGRAPH()
        end
    end

Subroutine PROCESSREFS(π, j, ChunkSize)
1   ValueSet = GETVALUESET(π, resourceLimit, j,
    ChunkSize)
2   for σ in ValueSet do
3       MP ← GETMOTIONPLAN(σ, π, resourceLimit)
4       if MP ≠ NULL then
5           return success
        end
    end
6   for σ in ValueSet do
7       LastReachableSt, FailedPrecon ← TRYGETMP(σ, π)
        /* Make last reachable state unsolvable
           if FailedPrecon not found.               */
8       TrueLRState ← PATCH(LastReachableSt,
        FailedPrecon)
9       /* Disallow all states up to and
           including LastReachableSt.               */
10      Dom2 ← DISALLOWHISTORY(Dom, π,
        LastReachableSt)
11      PROCESSSDH(TrueLRState, Dom2, π, σ)
    end

Subroutine PROCESSSDH(State, Domain, PlanHistory,
Interp)
1   SDHTuple = ⟨ State, Domain, PlanHistory, Interp ⟩
2   plan = GETCLASSICALPLAN(SDHTuple)
3   if plan == NULL then
        /* Disallow action which led to failure
           state from previous state.               */
4       State2, History2 ← GETPREDECESSORS(SDHTuple)
5       Dom2 ← DISALLOWACTIONINSTATE(Domain,
        State2, LASTACTION(History2))
6       PROCESSSDH(State2, Dom2, History2)
    end
7   else
        /* Add plan with edge from parent.          */
8       ADDTOPLANGRAPH(plan, HISTORY(SDHTuple))
    end
```

satisfy inverse kinematics constraints (IK feasibility). Trajectories are initialized as straight lines.

We call the MOTIONPLAN subroutine in line 3, 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 symbolic parame-

```
Algorithm RandRef(HLP, N_max)
1    init ← InitRefinement(HLP)
2    for iter = 0, 1, ..., N_max do
3        failStep, failPred ← MotionPlan(HLP)
4        if failStep == NULL then
             /* Found valid plan refinement.      */
5            return success
         end
6        else if failPred == NULL then
             /* Motion planning failure.          */
7            failAction ← HLP.ops[failStep]
8            Resample(failAction.params)
         end
9        else
             /* Action precondition violation.    */
10           Resample(failPred.params)
         end
     end
11   Raise failure to task planner, receive new plan.
```

ters associated with a failure; this routine picks one of these at random and resamples its value. If we reach the iteration limit (l.11), 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.

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 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 symbolic references.

VI. Learning Refinement Distributions

In this section, we present an RL approach that learns a policy for plan refinement. We show how to train continuous proposal distributions, thus avoiding the lack of robustness from hand-coded discretizations in other TAMP systems.

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 setting of values for symbolic references, 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 symbolic reference 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 reference *type*, comprised of a pose type and possibly a gripper (e.g., "left gripper grasp pose," "right gripper putdown pose," "base pose"). 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 $s \in \mathcal{S}$, symbolic reference 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 symbolic reference p with sample value x, in state $s \in \mathcal{S}$:

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

We define the expected reward of an episode ξ:

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

and approximate 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 [12]. 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.

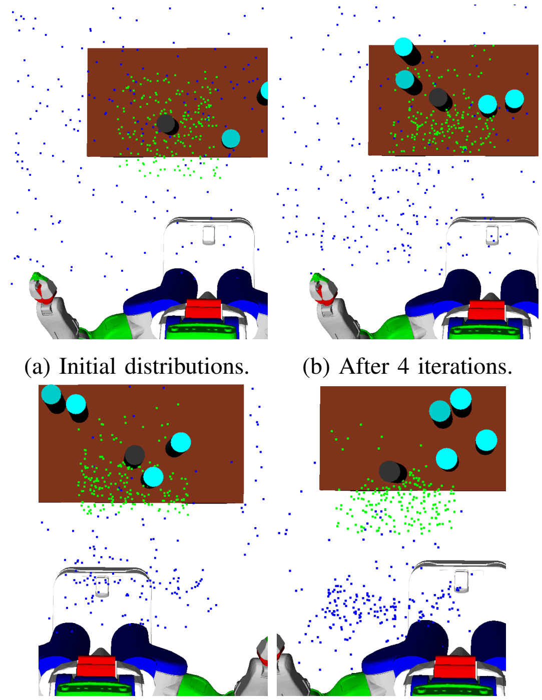
VII. Learning to Search Plan Refinement Graph

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, thus constituting a meta-level search. Section IV-B 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 symbolic references.

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 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.

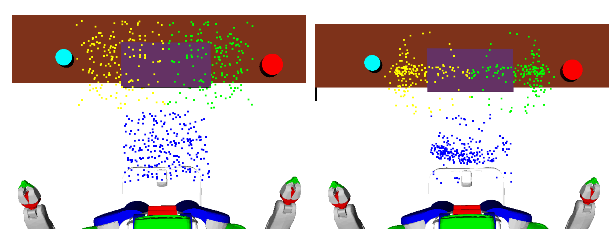
The features for our regressors are as follows. 3 geometric features encode the closeness of the objects of interest in our environment, considering the distance to and placement of nearby obstructions. The other feature describes how many times the node has been visited before.



(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 $\vec{\mathbf{0}}$. The final distributions are after 12 iterations.



(a) Initial distributions. (b) Final distributions.

Fig. 3: Initial and learned base position (blue) and tray pickup (green, yellow) distributions. The green points refer to where the right gripper will be placed; the left gripper is placed in a symmetric position on the other side of the tray, as marked by the yellow points. The initial distributions are uniform because we initialize weights to $\vec{\mathbf{0}}$. The final distributions are after 20 iterations.

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.

VIII. Experiments

A. Training Methodology

We use the reward function described earlier. Our weight vectors are initialized to $\vec{\mathbf{0}}$ for all plan parameter types – this initialization represents a uniform distribution across the

# Objects	System	% Solved (SD)	Avg Ref Time (s)	Avg # MP Calls
2 (dinner)	T	100 (0)	35.5	60.2
2 (dinner)	B	100 (0)	37.3	59.2
2 (dinner)	L	99 (1.8)	41.5	61.6
4 (dinner)	T	100 (0)	43.2	98.0
4 (dinner)	B	90 (0)	63.0	95.5
4 (dinner)	L	99 (0.6)	69.2	97.1
25 (cans)	T	74 (0)	5.4	12.7
25 (cans)	B	76 (0)	9.7	9.7
25 (cans)	L	92 (4.6)	12.7	10.1
25 (cans)	F	95 (2.7)	9.2	11.6
30 (cans)	T	42 (0)	6.7	9.2
30 (cans)	B	42 (0)	27.6	15.9
30 (cans)	L	72 (9.8)	18.5	12.6
30 (cans)	F	83 (3.0)	10.2	12.8

TABLE I: Percent solved and standard deviation, along with time spent refining and number of calls to the motion planner for baseline 1 (T), baseline 2 (B), our learned refinement policies with the graph search used in baseline 2 (L), and our full system: learned refinement policies and graph search heuristics (F). Results for L and F are averaged across 10 separately trained sets of weights. Time limit: 300s.

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 reference 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 two baselines, both of which use the hand-coded sampling discretizations for plan refinement used in SFRCRA-14. Baseline 1 is SFRCRA-14: it uses exhaustive backtracking search for refinement and greedy depth-first search of the plan refinement graph, which always tries to refine the plan that incorporates all error information obtained thus far. Baseline 2 uses randomized refinement with the following fixed graph search policy: try 3 times to refine the deepest node in the graph; if unsuccessful, generate a geometric fact from it, replan (which creates a child node), and repeat.

For the can domain, we report results for 4 systems: 1) baseline 1; 2) baseline 2; 3) our learned refinement policies with the graph search used in baseline 2; and 4) our full system, with learned refinement policies and graph search heuristics. For the dinner domain, we report results only for the first 3 systems, because the errors propagated in this domain relate to the stackability of objects. Since this is independent of reference instantiations, we want to incorporate all available error information when attempting refinement. Thus, the graph search strategy from baseline 2 can be expected to perform well in this setting.

For the fourth 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 third 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 third and fourth 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 [13] with a PR2 robot. The motion planner we use is trajopt [14], and the task planner is Fast-Forward [15]. The experiments were carried out in series on an Intel Core i7-4790K machine with 16GB RAM. Table I summarizes our quantitative results. Figure 1 and Figure 2 show learned refinement policies for the can domain. Figure 3 shows learned tray pickup poses for the dinner domain.

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 references: 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 reference types for $N = 30$ iterations with $\epsilon = 20$. We fixed $L = 100$.

The results demonstrate significant improvements in performance to the baseline systems for success rate. However, backtracking search provides faster average refinement time. This is likely because the refinement times were averaged over the test cases where all 4 systems succeeded. These plans tended to be easier to refine, so an exhaustive backtracking search performs well because the total search space is small.

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 references: 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 systems. 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 systems' discretization. Additionally, the lack of long-term dependencies in the plan means that backtracking search finds a valid refinement quickly. The fact that our system performs comparably with the baselines shows that our algorithm can learn grasp poses for a variety of objects, such as cylinders and a tray.

IX. Conclusion

We presented a complete algorithm for TAMP. We then applied machine learning to train continuous proposal distributions for plan refinement and search intelligently through a plan refinement graph. Thus, our full system learns both *which* node to refine and *how* to perform the refinement. We evaluated performance against SFRCRA-14 in 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] 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] S. Chib and E. Greenberg, "Understanding the metropolis-hastings algorithm," *The american statistician*, vol. 49, no. 4, pp. 327–335, 1995.

[13] 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.

[14] 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.

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