Guided Search for Task and Motion Plans Using Learned Heuristics

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

***Abstract*— In mobile manipulation planning, it is not uncommon for tasks to require thousands of individual motions. Such problems require reasoning about courses of action from the viewpoint of logical objectives as well as the feasibility of individual movements in the configuration space. In discrete representations, planning complexity is exponential in the length of the plan; in mobile manipulation, the set of parameters for an action is often continuous, so we must also cope with an infinite branching factor. *Task and motion planning* (TAMP) methods integrate logical search with continuous geometric reasoning to address this challenge. We present an algorithm for searching in the space of possible task and motion plans. We develop novel techniques for using statistical machine learning to guide the search process. Our contributions are as follows: 1) we present a complete algorithm for TAMP; 2) we present a randomized local search algorithm for TAMP that is easily formulated as a Markov decision process (MDP); 3) we give a reinforcement learning (RL) algorithm that learns a policy for this MDP; 4) we present a method that trains heuristics for intelligently searching the available space of task plans, given options that address different infeasibilities; and 5) we run experiments to evaluate the performance of our system in a variety of simulated domains. We show significant improvements in performance 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]. 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.

In this work, we develop a complete algorithm for TAMP and propose methods for carrying out guided search in the space of high-level (logic-based) plans and their low-level *refinements*, or instantiations of continuous values for symbolic references in the plan.

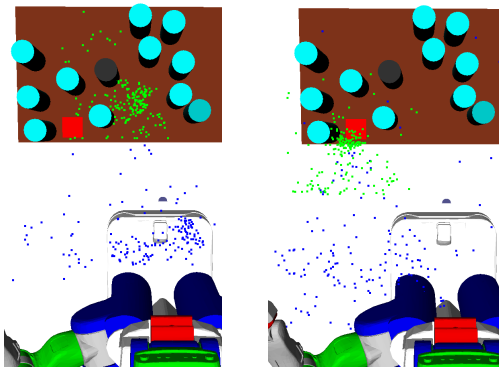


Fig. 1: Screenshots showing distributions learned by our system in a simulated pick-and-place domain. We use reinforcement learning to train good sampling distributions for continuous motion planning parameters in long-horizon tasks. The robot must grasp the black can and put it down on the red square.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

In particular, we introduce a *plan refinement graph*, which allows interleaving plan refinement (the search for symbolic reference instantiations of a plan) with a search over *which* high-level 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, making them more efficient than previous hand-coded approaches. For example, the TAMP systems cited above use hand-coded heuristics to search for instantiations of symbolic references, relying on domain-specific insight from the user to reduce the space of possible values.

To learn efficient plan refinement strategies, we train distributions to propose continuous values for symbolic references that are likely to yield successful trajectories using a motion planner. We train these distributions through an application of reinforcement learning (RL). Our approach draws inspiration from Zhang and Dietterich [6], who applied RL to job shop scheduling. In their formulation, states correspond to schedules and actions propose changes to the schedule. In our setting, states correspond to (potentially infeasible) refinements and actions propose new values for symbolic references. We implement our approach using methods adapted from Zucker et al. [7], who train a configuration space sampler for a randomized motion planner.

The contributions of our work are as follows: 1) we

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

present a complete algorithm for TAMP by maintaining a plan refinement graph; 2) we present 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.

Xu et al. [9] 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 defined as 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. The solution to a motion planning problem is a collision-free trajectory that connects p_0 and p_t. To allow for object manipulation, we let $\mathcal{X}$ include poses for each movable object in the environment.

In task and motion planning, we add more abstract concepts to this formulation, including *fluents* (logical properties that hold either true or false and may vary across configurations) and *actions* (operations that the agent may choose to execute, resulting in changes to the configuration and the 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 to achieve a desired goal condition expressed in terms of fluents.

For example, we can use the action schema *grasp(Object o, Manipulator p, GraspingPose g, Trajectory m)* to abstractly represent grasping an object, *o*. In order to apply this action, the agent must select values for each of these parameters (e.g., *grasp(can*$_1$*, left,* g_1*,* m_1*)*). These *instantiated* actions change the value of specific fluent instantiations (also called fluent *literals*), such as *in-gripper(can*$_1$*, gripper*$_l$*)* and *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, trajectories, and poses. Note that $\mathcal{O}$ includes the configuration space of all movable objects, including the robot.
- $\mathcal{T}$ is a set of object *types*, such as movable objects, 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 if and only if its preconditions hold in that state. Implicitly, the trajectories corresponding to actions 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) provide a way to formalize 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, $\pi : \mathcal{S} \rightarrow \mathcal{A}$, that maps states to actions. The value, $V_\pi(s)$, of a state under π is the sum of expected discounted future rewards generated 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 [10].

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 $TD(\lambda)$ [10] 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 adapt their gradient updates to the TAMP framework (Section V-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

SFRCRA-14 solves TAMP problems by: incrementally searching for a high-level plan that solves the logical 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. This search process addresses the fundamental TAMP problem: high-level logical descriptions are lossy abstractions of the true environment dynamics and thus may not include sufficient information to determine the true applicability of a sequence of actions.

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 *high-level*, or *symbolic*, 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 for symbolic references while ensuring action feasibility. 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, this algorithm has two main limitations: it is not guaranteed to find a solution when there exists one, and the sets of values from which instantiations get sampled are object-specific, hand-coded distributions. Since the algorithm never reduces the set of possible sampled values, its efficiency degrades as the number of values that get sampled increases. In the next subsection, we address the first limitation; in the following sections, we address the second.

B. A Complete Algorithm for TAMP

We introduce a complete algorithm that maintains a *plan refinement graph* (PRGraph). Every node u in the PRGraph represents a high-level plan π_u and the current state of the search for a refinement. An edge (u, v) in the PRGraph represents a "correction" of π_u for a specific instantiation of the symbolic references in π_u. Let $\pi_{u,k}$ be the plan

prefix of π_u consisting of the first k actions. Formally, each edge $e = (u, v)$ is labeled with a tuple $\langle \sigma, k, \varphi \rangle$. σ denotes an instantiation of references for a prefix $\pi_{u,k}$ of π_u such that feasible motion plans have been found for all previous actions $\pi_{u,k-1}$. φ denotes a conjunctive formula consisting of fluent literals that were required in the preconditions of the k^{th} action in π_u but were not true in the state obtained upon application of $\pi_{u,k-1}$ with the instantiation σ_k. The plan in node v (if any) retains the prefix $\pi_{u,k-1}$ and solves the new high-level problem which incorporates the discovered facts $\varphi_{u,v}$ in the k^{th} state.

The overall search algorithm interleaves the search for feasible refinements of each high-level plan with the addition into the PRGraph of new edges and plan nodes using the semantics described above. This process is described using non-deterministic choices (denoted using the prefix "ND") in Alg. 1. Subroutine REFINENODE selects a reference instantiation and attempts to solve the motion planning problems corresponding to it; subroutine ADDCHILD selects a reference instantiation and creates a new node that either incorporates the reason of infeasibility (provided by the domain-specific subroutine GETERROR), or makes a random change in the high-level plan. The latter can be required in some pathological domains that have dead-ends and where changing the instantiation of symbolic references for an action has no effect on the action outcomes.

Different implementations of the non-deterministic choices in Alg. 1 can capture various search algorithms with adaptations for handling unbounded branching factors (e.g., iterative-deepening with iterative-broadening best first search). Indeed, SFRCRA-14 can be seen as a greedy depth-first traversal of the PRGraph. We will show that using trained guided search heuristics with the PRGraph can lead to performance improvements.

It is easy to see that the resulting algorithm is complete.

Theorem 1: If there exists a high-level sequence of actions that a) does not revisit symbolic states when using the high-level domain definition and b) has a motion planning feasible refinement within the scope of symbol interpretations, then Alg. 1 will find it.

The proof follows easily because if there is a solution, then the non-deterministic calls can be selected appropriately to find it. In the next section, we show a specific implementation of REFINENODE based on randomization. Afterward, we show how to train heuristics that guide the search processes, replacing the non-deterministic choices.

C. A Randomized Algorithm for Plan Refinement

In order to apply the complete planning algorithm described above, we must provide definitions for each of the subroutines mentioned in Alg. 1. There are many ways to implement these functions. For example, SFCRA-14 uses a backtracking search over a discrete set of instantiations to implement REFINENODE. Since we want to apply RL to learn policies for refinement, we seek an algorithm that allows for easy formulation as an MDP. Our method imitates that of Zhang and Dietterich [6]: we initialize an infeasible

```
   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        u ← NDGETNEXTNODE(PRGraph)
4        mode ← NDChoice{refine, add child}
5        if mode == refine then
           REFINENODE(u, j)
         else
           ADDCHILD(u, j)
         end
       end
     end

     Subroutine REFINENODE(u, j)
1      π ← HLPlan(u)
2      for j in 1 .. N_max do
3        σ ← NDGETINSTANTIATION(π, j)
           /* resourceLimit(j) is
              monotonically increasing in j          */
4        MP, FailedAction, FailedPred ←
         GETMOTIONPLAN(σ, π, resourceLimit(j))
5        if MP ≠ NULL then
6          return success
         end
       end


     Subroutine ADDCHILD(u, j)
1      π ← HLPlan(u)
2      σ ← NDGETINSTANTIATION(π, j)
3      StepNum, FailedPrecon ← GETERROR(σ, π)
4      mode ← NDChoice{error, random}
5      if mode == error then
6        newState ← PATCH(getStateAt(StepNum, π),
         FailedPrecon)
       else
7        π ← π, with an action before StepNum replaced
                 by a random applicable action
8        newState ← getStateAt(stepNum, π)
       end
9    π′ ← GETCLASSICALPLAN(newState)
10   ADDNODETOPRGRAPH(σ, stepNum, π′)
```

Algorithm 1: Complete algorithm for TAMP.

refinement and use a randomized local search to propose improvements. Alg. 2 shows pseudocode for this refinement strategy, which implements REFINENODE in Alg. 1. The main difference between our implementation and the pseudocode in Alg. 1 is that we use information about previous failed motion planning attempts to guide selection of the next instantiation.

The algorithm takes as input a high-level plan and a maximum iteration count. In line 1, we initialize a (potentially invalid) refinement by sampling from distributions associated with each symbolic reference. We continue sampling until we find bindings that satisfy inverse kinematics constraints (IK feasibility). Trajectories are initialized as straight lines.

The MOTIONPLAN subroutine called in line 3 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

Algorithm *RandRef*(HLP, N_{max})

```
init ← INITREFINEMENT(HLP)
for iter = 0, 1, ..., N_max do
    failStep, failPred ← MOTIONPLAN(HLP)
    if failStep == NULL then
        /* Found valid plan refinement.      */
        return success
    end
    else if failPred == NULL then
        /* Motion planning failure.          */
        failAction ← HLP.ops[failStep]
        RESAMPLE(failAction.params)
    end
    else
        /* Action precondition violation.    */
        RESAMPLE(failPred.params)
    end
end
```

Algorithm 2: Randomized local search for plan refinement.

the sampled poses. If this succeeds, it tests the action preconditions; as part of this step, it checks that the trajectory is collision-free.

We then call the RESAMPLE routine on the symbolic parameters associated with the infeasibility; this routine picks one at random and resamples its value. INITREFINEMENT and RESAMPLE together define NDGETINSTANTIATION for our implementation, while GETERROR iterates through the steps of the plan, checks precondition and trajectory feasibility, and returns a failed action index and associated predicate.

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. This allows plan refinement to account for long-term dependencies in the instantiation of symbolic references.

V. LEARNING REFINEMENT DISTRIBUTIONS

Randomized refinement provides us with a framework to learn a policy that implements NDGETINSTANTIATION and performs well empirically. Our approach applies RL to train continuous proposal distributions for symbolic reference instantiations in this routine.

A. Formulation as Reinforcement Learning Problem

We formulate plan refinement as an MDP as follows:

- A state $s \in \mathcal{S}$ is a tuple (HLP, r_{cur}, E, n), consisting of the high-level plan, its current setting of values for symbolic references, the geometric environment encoding, 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, encompassing both the task planning problem and the environment.
  - 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.

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 problem instances.

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, after which we update weights.

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 [11]. 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.

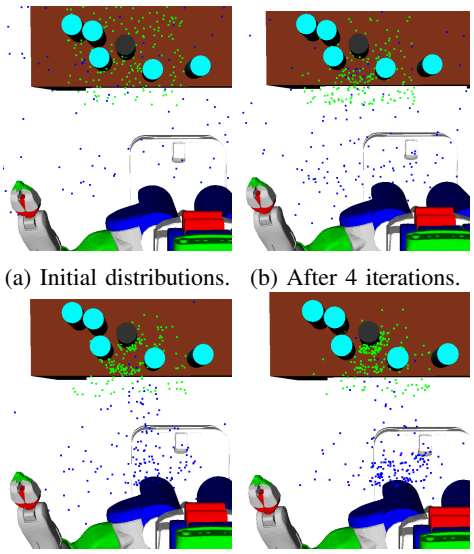
VI. Learning to Search the Plan Refinement Graph

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. Recall that in Alg. 1, the high level has a two-tiered decision to make: which node in the plan refinement graph to visit next, and whether to attempt to refine this node or generate failure information from it. These decisions are encoded in the routines NDGetNextNode and NDChoice in line 4 of the main algorithm. In this section, we show how to train heuristics that implement these routines. The heuristics are trained to estimate the difficulty associated with refining a plan, and accordingly decide to generate a geometric fact to use for replanning.

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.

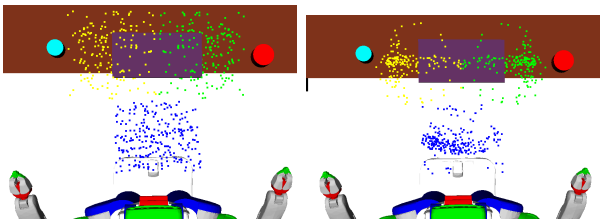
To train the regressors, we fix pre-trained policies for plan refinement and construct datasets for supervised learning as follows. For the first regressor, we run refinement on the root node of the graph over 500 random environments sampled from $\mathcal{P}$, and measure the feature vector and number 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. Initial distributions are uniform because we initialize weights to $\vec{\mathbf{0}}$. 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 for the dinner domain. 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. 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. If refining a child node would reduce the number of steps to a valid refinement, we bias toward generating a child node.

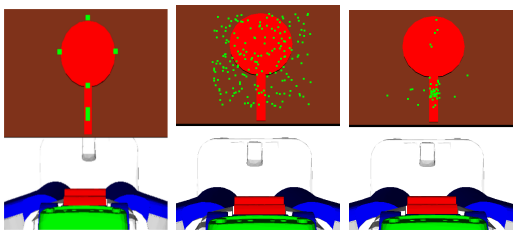
VII. Experiments

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 limits of the geometric search space. We use 24 features

# Objects	System	% Solved (SD)	Avg Ref Time (s)	Avg # MP Calls
25 (can)	T	80 (0)	5.1	9.6
25 (can)	B	80 (0)	12.6	10.3
25 (can)	L	93 (5.0)	12.0	11.1
25 (can)	F	95 (1.0)	10.6	10.6
30 (can)	T	42 (0)	7.2	8.7
30 (can)	B	42 (0)	19.6	11.1
30 (can)	L	77 (9.7)	13.1	12.8
30 (can)	F	86 (2.0)	11.2	12.8
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
2 (frying)	T	96 (0)	29.0	67.2
2 (frying)	B	88 (0)	46.9	60.0
2 (frying)	L	99 (2.0)	22.6	44.7
4 (frying)	T	55 (0)	48.9	131.8
4 (frying)	B	20 (0)	187.9	155.5
4 (frying)	L	92 (6.8)	90.6	120.9

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 5 separately trained sets of weights. As described, we only run F for the can domain. Time limit: 300s.



(a) Hand-coded distribution. (b) Initial distribution. (c) Final distribution.

Fig. 4: Hand-coded distribution, along with initial and final distributions using our training methods, for picking up the frying pan. The green points refer to where the gripper will be placed. Our system learned to prefer picking up the pan at its handle to fit it into the shelf (not shown). Final distributions are after 20 iterations.

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 three distinct domains: cans distributed on a table (the *can domain*), setting up bowls for dinner (the *dinner domain*), and placing frying pans into a narrow shelf (the *frying domain*). We compare performance with two baselines, both of which use the hand-coded sampling distributions for 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 and frying domain, we report results only for the first 3 systems, because the errors propagated in these domains relate to the stackability of objects. Since this is independent of the current refinement, 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 these settings.

We employ the following algorithm to produce a trained set of weights θ_p for refinement. We train 3 sets independently, test each one on a validation set, and output the best-performing one. We found that this reduced variation due to random seeding.

We report results on fixed test sets of 50 randomly generated environments for the can and dinner domains, and 20 for the frying domain (because the frying domain environments do not have as much variation). For the third and fourth systems, we average results across running the training process 5 times independently and evaluating each final set of weights.

Our experiments are conducted in Python 2.7 using the OpenRave simulator [12] with a PR2 robot. The motion planner we use is trajopt [13], and the task planner is Fast-Forward [14]. The experiments were carried out in series on an Intel Core i7-4770K machine with 16GB RAM. Table I summarizes our quantitative results.

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 refinement times were averaged over the test cases where all 4 systems succeeded. These plans tended to be easier to refine, so exhaustive backtracking search performs well because the total search space is small. Figure 1 and Figure 2 show learned refinement distributions.

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-coding the sample space works well 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 in the baseline systems. 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 pose instantiations for a variety of objects. Figure 3 shows learned tray pickup poses.

D. Frying Domain

We run two sets of experiments, using 2 and 4 frying pans. The robot must stack the frying pans in order of decreasing radius into a narrow shelf. To be successful, it must grasp the frying pans at the handle, so that the handle sticks out after the pan is placed in the shelf. This domain has 3 types of continuous references: base poses, pan grasp poses, and pan putdown poses. We do not use curriculum learning, as weights for all these parameters can be trained jointly. We fixed $N = 30$, $L = 100$, and $\epsilon = 5$. SFRCRA-14 did not have a frying domain, so we used the following hand-coded distribution for picking up the pans: 4 grasp poses in the cardinal directions around the lip of the pan, and 4 grasp poses equidistant along the handle.

The results demonstrate significantly higher success rate versus the baseline systems. The backtracking baseline is faster likely because the refinement times were averaged over cases where all 3 systems succeeded; backtracking often succeeded only when it "got lucky" and picked grasp poses along the handle early in the search. Figure 4 compares the hand-coded distribution with one we learned for picking up a frying pan.

VIII. 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] 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.

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

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

[12] 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.

[13] 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.

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