# 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*— Tasks in mobile manipulation planning often require thousands of individual motions to complete. Such tasks require reasoning about logical objectives as well as the feasibility of movements in configuration space. In discrete representations, planning complexity is exponential in the length of the plan; in mobile manipulation, parameters for an action often draw from a continuous space, so we must also cope with an infinite branching factor. *Task and motion planning* (TAMP) methods integrate a logical search over high-level actions with continuous geometric reasoning to address this challenge. We present an algorithm that searches the space of possible task and motion plans, 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 apply reinforcement learning (RL) to learn a policy for this MDP; 4) we learn from expert demonstrations to efficiently search the 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

We are interested in designing autonomous systems that can perform complex mobile manipulation tasks over long time horizons (e.g., setting a dinner table, doing laundry). We approach this problem in the framework of combined *task and motion planning* (TAMP).

In TAMP, an agent is given a symbolic, logical characterization of actions (e.g., move, grasp, putdown), along with a geometric encoding of the environment. Efficient integration of high-level, symbolic task planning and low-level, geometric motion planning is difficult; recent research has proposed several methods for it [1], [2], [3], [4], [5], [6]. 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 learning methods to perform joint guided search in the space of high-level, symbolic plans and their low-level *refinements*: instantiations of continuous values for symbolic references in the plan. For example, in a pick-place domain, a high-level plan consists of a sequence of grasp and putdown actions, while its refinement is a sequence of collision-free trajectories that implement the plan. We refer to this search for a valid refinement as *plan refinement*.

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

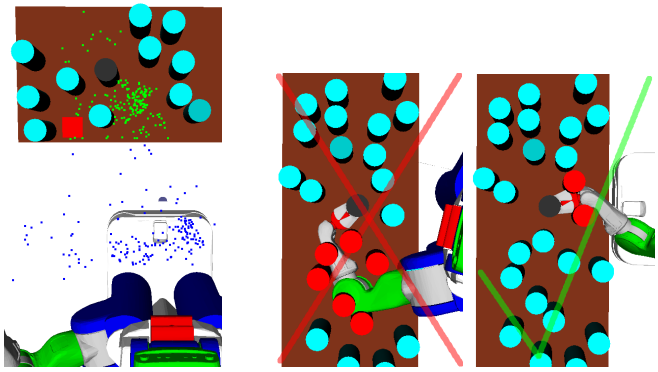


Fig. 1: **Left**: We use reinforcement learning to train good sampling distributions for continuous motion planning parameters in long-horizon tasks. The image shows learned base position (blue) and grasping (green) distributions for picking up the black can. The grasping policy learned to avoid the obstructions. The green points refer to the position of the tool center point; the end effector is oriented to point toward the object being grasped. **Right**: When trying to grasp the black can, the sampled grasping pose can significantly affect the quality of the obstructions determined. The new plan proposed from the rightmost image only requires moving 2 obstructions out of the way, versus 6 from the other. We integrate inverse reinforcement learning into our task and motion planning system to learn an ordering on plan exploration, causing the system to prefer simpler and more feasible plans.

SFRCRA-14 uses error information propagated from the geometric planner to update the symbolic state and generate a new high-level plan. For example, if motion planning discovers obstruction information in a pick-place domain, the new task plan may involve moving the obstruction out of the way. In our work, we think of these errors as defining a *plan refinement graph* whose nodes are high-level plans and edges are error fluents. We develop a complete algorithm that interleaves search over this graph with plan refinement.

We present machine learning techniques to train heuristic functions that guide search over both the high-level plan refinement graph and low-level plan refinement. Both search spaces are very large, so we need heuristics to make search efficient. At the low level, this amounts to learning to propose continuous values (for symbolic references) that are likely to result in collision-free trajectories. Many TAMP systems rely on hand-coded heuristics to account for these continuous parameter settings. This often requires domain-specific insight from the user to reduce the search space. We apply reinforcement learning (RL) to learn domain-specific distributions over these values in a domain-independent fashion. At the high level, training heuristics amounts to learning how difficult it is to refine a given plan. Directly applying reinforcement learning for this is challenging because taking

actions in this space requires attempting to refine a plan, which can be time-consuming. Also, for complex tasks, there are often many candidate high-level plans that achieve the goal. Instead, we use inverse reinforcement learning based on expert demonstrations to train heuristics at this level.

Our low-level learning approach draws inspiration from Zhang and Dietterich [7], 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.

The contributions of our work 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 an MDP; 3) we apply RL to learn a policy for this MDP; 4) we learn from expert demonstrations to efficiently search the space of high-level 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. Our results demonstrate significantly improved performance over SFRCRA-14.

## II. Related Work

Dearden and Burbridge [2] use machine learning for TAMP by learning probability models that map symbolic predicates to geometric states. They use these models in a backtracking search over potential geometric state instantiations for the sequence of symbolic states visited by the plan. Their work differs from ours in two key ways. First, they focus on learning the semantics of symbolic predicates, whereas we assume known semantics and instead optimize for fast planning. Second, they sample geometric states independently for each symbolic state in the task sequence, while our distributions are conditioned on the entire plan and its current refinement.

Kaelbling et al. [3] 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. [4] propose a set of geometric constraints that involve 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. By contrast, our approach learns good geometric values to propose based on experience.

Garrett et al. [5] use information about reachability in the configuration space and symbolic state space to construct a *relaxed plan graph* that guides motion planning, using geometric biases to break ties. Intermediate poses are sampled from a hand-coded distribution, whereas we learn these distributions using RL and could use them within their search.

Our problem formulation is motivated by Zhang and Dietterich's application of RL to job shop scheduling [7]. Job shop scheduling is a combinatorial optimization problem where the goal is to find a minimum-duration schedule of a set of jobs with temporal and resource constraints. An empirically successful approach to this problem relies on a randomized local search that proposes changes to an existing suboptimal schedule. The authors formulate this as an MDP and use TD($\lambda$) [8] with function approximation to learn a value function for it. Their approach outperforms the previous state of the art for this task and scales better to larger scheduling problems.

Zucker et al. [9] use RL to bias the distribution of a rapidly exploring random tree (RRT) for motion planning. They use 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).

Another line of research has been devoted to machine learning techniques that guide search algorithms. This general formulation is applied to many domains other than hierarchical planning for robotics. Boyan et al. [10] solve optimization problems by learning a state evaluation function that guides local search. Arbelaez et al. [11] solve constraint satisfaction problems using a heuristic model that is refined through supervised learning. Xu et al. [12] train heuristics for controlling forward state-space beam search in classical task planners.

## III. Background

### A. Task and Motion Planning

A motion planning problem is defined by a configuration space for a robot and all movable objects in its environment, along with initial and final configurations. The solution to a motion planning problem is a collision-free trajectory for the robot that connects these configurations. In task and motion planning, we add more abstract concepts to this formulation.

*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 $\mathcal{O}$ defines the configuration space of all movable objects, including the robot.
- $\mathcal{T}$ is a set of object *types*, such as movable objects, 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* parameterized by 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.

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

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 a finite-horizon MDP as a tuple $\langle \mathcal{S}, \mathcal{A}, T, R, H, \mathcal{P} \rangle$, where

- $\mathcal{S}$ is the state space.
- $\mathcal{A}$ is the action space.
- $T(s, a, s') = Pr(s' \mid s, a)$ 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.
- $H$ is the horizon, or total number of timesteps.
- $\mathcal{P}$ is the initial state distribution.

A solution to an MDP is a policy, $\pi : \mathcal{S} \times \mathbb{Z} \rightarrow \mathcal{A}$. The value function under $\pi$ is a function of the timestep $k$ and state $s$:

$$V_{\pi}^{k}(s) = \mathbb{E}\left[\sum_{t=k}^{H} R(s_t) \mid \pi, s_k = s\right].$$

The optimal policy, $\pi^*$, is time-varying and maximizes $V$.

In reinforcement learning (RL), an agent must determine $\pi^*$ 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; standard techniques include value function approximation, which uses methods such as temporal difference learning, and direct policy estimation, which encompasses gradient-based and gradient-free methods [8].

In inverse RL [13], an agent must recover $R$ given execution traces of optimal behavior according to $\pi^*$. This is useful in apprenticeship learning settings where the agent must acquire skills from observing an optimal demonstrator. Standard techniques include the maximum-margin method [14] and the maximum-entropy method [15].

## 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

The fundamental TAMP problem is that high-level logical descriptions are lossy abstractions of the true environment dynamics. Thus, they may not include sufficient information to determine the applicability of a sequence of actions. SFRCRA-14 addresses this 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.

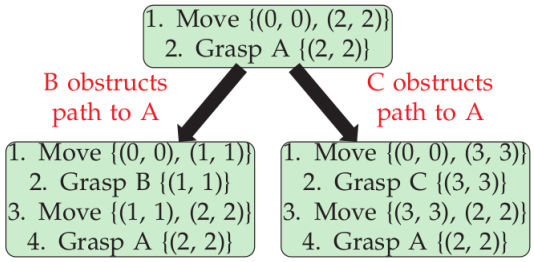



Fig. 2: A simple plan refinement graph for an environment with 3 objects: A, B, and C. The goal is to grasp object A. Each node maintains a high-level plan and a set of instantiated continuous values for its symbolic references. The edges are labeled with errors discovered by the low-level motion planner; these errors are propagated back up to the task planner for replanning.

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 is 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* (PRG). Figure 2 illustrates a simple example with 3 high-level plans. Every node $u$ in the PRG represents a high-level plan $\pi_u$ and the current state of the search for a refinement. An edge $(u, v)$ in the PRG represents a "correction" of $\pi_u$ for a specific instantiation of the symbolic

references in $\pi_u$. Let $\pi_{u,k}$ be the plan prefix of $\pi_u$ consisting of the first $k$ actions. Formally, each edge $e = (u, v)$ is labeled with a tuple $\langle \sigma, k, \varphi \rangle$. $\sigma$ denotes an instantiation of references for a prefix $\pi_{u,k}$ of $\pi_u$ such that feasible motion plans have been found for all previous actions $\pi_{u,k-1}$. $\varphi$ denotes a conjunctive formula consisting of fluent literals that were required in the preconditions of the $k^{th}$ action in $\pi_u$ but were not true in the state obtained upon application of $\pi_{u,k-1}$ with the instantiation $\sigma_k$. 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 of new edges and plan nodes into the PRG 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 1) incorporates the reason for infeasibility (provided by the subroutine GETERROR), or 2) holds a nearly identical high-level plan, but with a random change at a single step. 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. GETERROR returns some failed precondition for an infeasible refinement; a typical implementation may, for example, run collision checking on the current set of trajectories to detect obstruction information.

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 PRG. We will show that using trained guided search heuristics with the PRG 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, as long as the non-deterministic policies assign non-zero weight to each choice.

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, SFRCRA-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 [7]: we initialize an infeasible 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 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 comprising 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

```
   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.                    */
3        u ← NDGETNEXTNODE(PRG)
4        π ← GETHLPLAN(u)
5        mode ← NDCHOICE{refine, add child}
6        if mode == refine then
           | REFINENODE(π, j)
         else
           | ADDCHILD(π, j)
         end
       end
     end

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


     Subroutine ADDCHILD(π, j)
1      σ ← NDGETINSTANTIATION(π, j)
2      StepNum, FailedPrecon ← GETERROR(σ, π)
3      mode ← NDCHOICE{error, random}
4      if mode == error then
5        NewState ← PATCH(GETSTATEAT(StepNum, π),
         FailedPrecon)
       else
6        π ← π, with an action before StepNum replaced
                by a random applicable action
7        NewState ← GETSTATEAT(StepNum, π)
       end
8      π′ ← GETCLASSICALPLAN(NewState)
9      ADDNODETOPRG(σ, StepNum, π′)
```

**Algorithm 1:** Complete algorithm for TAMP.

**Algorithm** *RandRef*($\pi, N_{max}$)

$\sigma \leftarrow$ INITREFINEMENT($\pi$)
**for** *iter = 0, 1, ...,* $N_{max}$ **do**
  $failStep, failPred \leftarrow$GETMOTIONPLAN($\pi$)
  **if** $failStep$ == *NULL* **then**
    /* Found valid plan refinement. */
    return success
  **end**
  **else if** $failPred$ == *NULL* **then**
    /* Motion planning failure. */
    $failAction \leftarrow \pi.ops[failStep]$
    RESAMPLE($failAction.params$)
  **end**
  **else**
    /* Action precondition violation. */
    RESAMPLE($failPred.params$)
  **end**
**end**

**Algorithm 2:** Randomized local search for plan refinement.

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. 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. We apply RL to train continuous proposal distributions for symbolic reference instantiations.

### A. Formulation as Markov Decision Process

We formulate plan refinement as the following MDP:

- States are tuples $(\pi, \sigma, E)$ that consist of the high-level plan, its current (potentially infeasible) refinement, and the geometric environment.
- Actions are pairs $(p, x)$, 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 into 2 cases.
  - Case 1: The proposed value $x$ is IK infeasible. The state remains the same.
  - Case 2: Otherwise, the value of $p$ is set to $x$ and the motion planner is called.

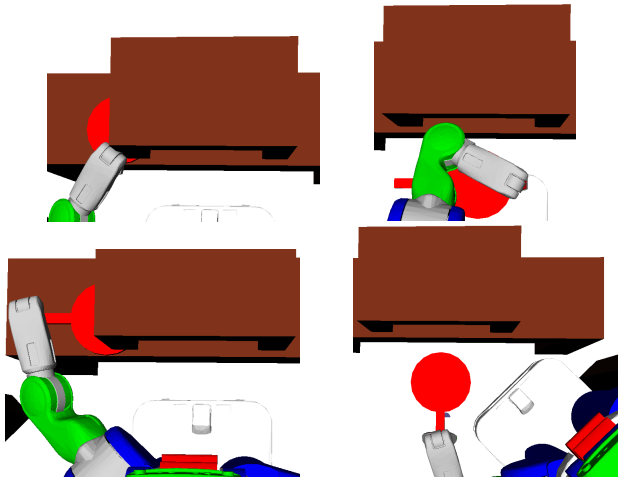


Fig. 3: **Top**: When the frying pan is not grasped by the handle, any attempted putdown pose for placing it into the narrow shelf fails. **Bottom**: When the frying pan is grasped by the handle, some putdown poses may succeed, as in the right, and some may fail, as in the left. In general, delayed rewards are important in the plan refinement MDP; sometimes multiple symbolic references require resampling to change an infeasible refinement into a feasible one.

- The reward function $R(s, a, s')$ provides rewards based on a measure of closeness to a valid plan refinement.
- The process terminates after $H$ timesteps.
- $\mathcal{P}$ is a distribution over planning problems and defines the initial state distribution for this MDP.

We restrict our attention to training policies that suggest $x$ for actions in $\mathcal{A}$, since randomized refinement already 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.

### B. Training Process

We learn a policy for this MDP by adapting the method of Zucker et al. [9], which uses a linear combination of features to define a distribution over poses. In our setting, we learn a weight vector $\theta_p$ 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 $f(s, a) = f(s, p, x)$ that maps the current state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$ 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 $\epsilon$ 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 $\Pi$. For each $\Pi$, run the randomized refinement algorithm to attempt to find a valid plan refinement, allowing the RESAMPLE routine to be called $H$ times before termination. Select actions according to the

$\theta_p$ and collect rewards according to $R$. After every $\epsilon$ calls to RESAMPLE, perform a gradient update on the weights.

Figure 3 shows that delayed rewards are an important aspect of the MDP: if a grasping pose along the lip of the frying pan is sampled, no putdown pose can lead to it being placed inside the shelf. Typically, an infeasible refinement must undergo resampling for several distinct symbolic references in order to reach a feasible refinement.

### C. Distribution and Gradient Updates

We adopt the sampling distribution used in Zucker et al. [9] 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 $\xi$:

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

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 $\xi$, and $\mathbb{E}_{q,s}[f]$ is the expected feature vector under $q$ in state $s$. The weight vector update is then:

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

for appropriate step size $\alpha$.

We sample $x$ from $q$ using the Metropolis algorithm [16]. Since our distributions are continuous, we cannot easily calculate $\mathbb{E}_q[f]$, so we approximate it with a Monte Carlo estimate.

## VI. LEARNING WHAT TASK PLAN TO REFINE

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. Figure 1 illustrates why making this decision intelligently is critical to good performance. We now explain our inverse RL approach to training heuristics that implement these routines. The heuristics are trained to estimate the difficulty associated with refining a plan, and to decide when to quickly generate a geometric fact used for replanning with the task planner. In our explanation, $n$ is the selected node and $m$ is the mode to apply (either trying to refine the node or quickly generating failure information).

We exploit properties of the environment to hand-design a feature vector $f(n)$ that encodes the refinability of a single high-level plan. Because the mode $m$ is binary, we construct

$$f((n, m)) = \begin{bmatrix} f(n) \\ f(n) \end{bmatrix}^\top \begin{bmatrix} 1-m \\ m \end{bmatrix},$$

which stacks the feature vector for $n$ on itself, then turns off the bottom half when $m = 0$ and the top half when $m = 1$. Now that we have defined a feature vector associated with each decision in a given PRG, we can obtain human-demonstrated trajectories (sequences of actions $(n, m)^*$) that intelligently navigate the PRG. We then solve the following max-margin optimization problem with a constant margin and slack variables:

$$\begin{aligned} \min_{w, \xi_i \geq 0} \quad & ||w||^2 + C \sum_i \xi_i \\ \text{s.t.} \quad & w^\top f_i^* \geq w^\top f_{ij} + 1 - \xi_i \ \forall i, j \ , \end{aligned}$$

where the $i$ iterate over the demonstrated trajectories and the $j$ iterate over possible actions. Each demonstrated trajectory has an associated slack variable in this formulation. The weight vector $w$ encodes a ranking function on the different actions. We then use DAGGER [17] to augment our training data. At test time, we follow the policy encoded by $w$, picking the highest-scoring action at each step.

## VII. EXPERIMENTS

### A. Methodology

We evaluate our approach in three distinct domains: cans distributed randomly 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 refinement distributions 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 with the task planner (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.

For refinement distributions, initial experimentation revealed that training weights for all reference types jointly is intractable, because planning takes a long time. So, we apply curriculum learning by training with a planning problem distribution $\mathcal{P}$ that gets progressively harder. The details of this curriculum are described in each experiment's section. Additionally, we train the refinement policies first, then fix them while collecting demonstrations and training the graph

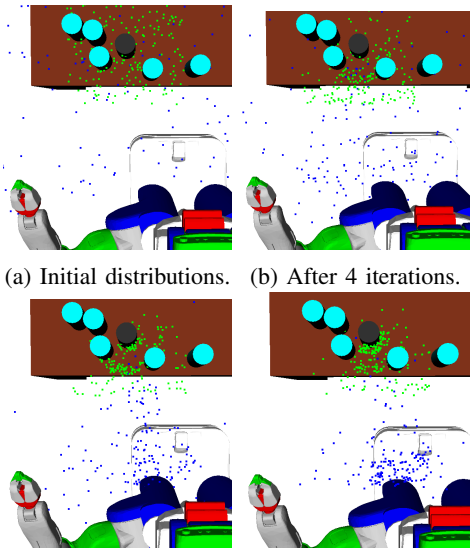
| # Objects | System | % Solved (SD) | Avg Ref Time (s) | Avg # MP Calls |
|---|---|---|---|---|
| 30 (can) | T | 42 (0) | 6.2 | 8.0 |
| 30 (can) | B | 40 (0) | 20.5 | 10.5 |
| 30 (can) | L | 72 (8.2) | 20.4 | 11.3 |
| 30 (can) | F | 81 (3.0) | 17.9 | 12.7 |
| 35 (can) | T | 50 (0) | 9.2 | 8.0 |
| 35 (can) | B | 50 (0) | 17.6 | 9.2 |
| 35 (can) | L | 68 (8.3) | 11.6 | 6.6 |
| 35 (can) | F | 78 (2.2) | 10.6 | 6.8 |
| 40 (can) | T | 34 (0) | 19.7 | 10.3 |
| 40 (can) | B | 36 (0) | 21.7 | 10.0 |
| 40 (can) | L | 61 (6.3) | 18.7 | 9.4 |
| 40 (can) | F | 74 (3.2) | 20.7 | 10.4 |
| 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.

search heuristics. For the fourth system, we reduce variance in the learning as follows. We train 3 sets of refinement weights independently, test each one on a validation set, and output the best-performing one. We set $\alpha = 0.1$.

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 these environments have less 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.

For training refinement distributions, our weight vectors $\theta_p$ are initialized to $\vec{\mathbf{0}}$ for each symbolic reference type. We use 24 features. 9 binary features encode the bucketed distance between the sample and target object. 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 $3\pi/4$.



(a) Initial distributions. (b) After 4 iterations. (c) After 8 iterations. (d) Final distributions.

Fig. 4: 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 $H$ calls to the RESAMPLE routine. Initial distributions are uniform because we initialize weights to $\vec{\mathbf{0}}$. Final distributions are after 12 iterations.

For training graph search, we use the following feature vector $f(n)$ associated with a high-level plan. Because the plan is composed of a sequence of object grasp and putdown actions, we first consider features of a single grasp action, targeted at an object $o$ in the environment. Consider a cone ranging from angles $-\frac{\pi}{3}$ to $\frac{\pi}{3}$ toward the closest table edge from $o$. The first feature, exists_obstr, is a binary variable indicating whether any other objects lie in this cone. The second, exists_path, is a binary variable indicating whether there is a linear grasp path wide enough for the robot's gripper to fit through within the cone. For the third feature, we approximate the robot's arm and gripper with a cylinder $c$ and sweep it across 10 discretized angles from $-\frac{\pi}{3}$ to $\frac{\pi}{3}$. We then store the minimum number of collisions with $c$ as the feature, sweep_count. This gives us a coarse approximation for the minimum number of object that should be moved before the target object is accessible via a linear path. We construct these features for the first five grasp actions in the plan (padding with -1 if there are not enough). We then add on the following aggregate features associated with the entire plan: 1) the minimum exists_obstr across all grasp actions, 2) the sum of sweep_count across all grasp actions, 3) the number of times $n$ was picked for refinement, and 4) the number of times $n$ was picked for generating an error.

Our experiments are conducted in Python 2.7 using the OpenRave simulator [18] with a PR2 robot. The motion planner we use is trajopt [19], and the task planner is Fast-Forward [20]. The experiments were carried out in series on an Intel Core i7-4770K machine with 16GB RAM. Table I summarizes our quantitative results.

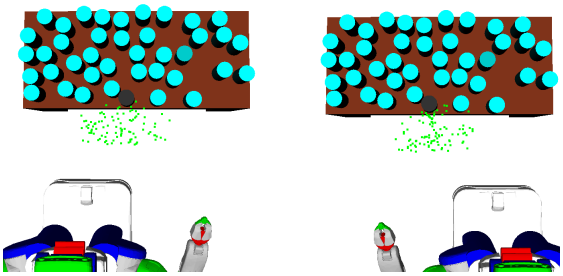
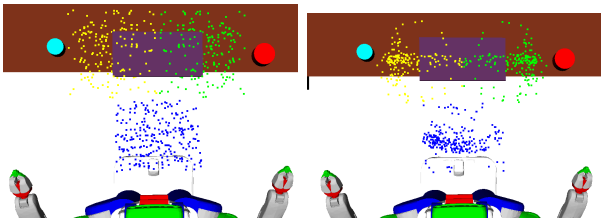


Fig. 5: The learned left arm grasp (green) distribution is affected slightly by relative robot location.



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

Fig. 6: Initial and learned base position (blue) and tray pickup (green, yellow) distributions for the dinner domain. The green points refer to the position of the right tool center point; the gripper is oriented toward the tray. 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.

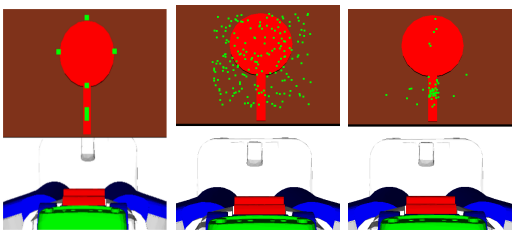
### B. Can Domain

We run three sets of experiments, using 30, 35, and 40 cans on the table. The goal is always for the robot to pick up a particular can 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.

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 $H = 100$.

To train the graph search heuristics, we collect approximately 300 optimal actions from the human demonstrator, over 3 rounds of DAGGER. After these 3 rounds, we found that performance plateaued. We use $C = 10^9$ in solving the max-margin optimization problem.

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 4 show learned refinement distributions. Figure 5 shows that our system learned to shape distributions based on the context in which an action is performed.



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

Fig. 7: Hand-coded distribution, along with initial and final distributions using our training methods, for picking up the frying pan. The green points refer to the position of the tool center point; the end effector is oriented downward. 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.

### 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 $H = 100$ 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 learning algorithm can recover good hand-coded distributions. Figure 6 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 $N = 30$, $H = 100$, 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 7 compares the hand-coded distribution with one we learned for picking up a frying pan.

## VIII. Limitations and Future Work

A major limitation of our system is the hand-coded features. In future work, we plan to move toward learned features, which offer more complex policy classes. Along the same lines, we hope to consider features of the logical structure of the high-level plan, perhaps using a kernelized method that applies to strings. We also plan to experiment with more sample-efficient RL algorithms.

Another limitation is the lack of variation among samples in the learned refinement distributions; we observed that similar samples were often selected repeatedly. To address this, we could incentivize exploring new regions of the sample space during training.

## 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] R. Dearden and C. Burbridge, "An approach for efficient planning of robotic manipulation tasks," *International Conference on Automated Planning and Scheduling*, 2013.

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

[4] F. Lagriffoul, D. Dimitrov, J. Bidot, A. Saffiotti, and L. Karlsson, "Efficiently combining task and motion planning using geometric constraints," *IEEE Conference on Robotics and Automation*, 2014.

[5] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "FFRob: An efficient heuristic for task and motion planning," *International Workshop on the Algorithmic Foundations of Robotics*, 2014.

[6] 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.

[7] W. Zhang and T. G. Dietterich, "A reinforcement learning approach to job-shop scheduling," *International Joint Conference on Artificial Intelligence*, 1995.

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

[9] M. Zucker, J. Kuffner, and J. A. D. Bagnell, "Adaptive workspace biasing for sampling based planners," *IEEE Conference on Robotics and Automation*, 2008.

[10] J. Boyan and A. W. Moore, "Learning evaluation functions to improve optimization by local search," *Journal of Machine Learning Research*, 2001.

[11] A. Arbelaez, Y. Hamadi, and M. Sebag, "Continuous search in constraint programming," in *Autonomous Search*, 2012, pp. 219–243.

[12] Y. Xu, S. Yoon, and A. Fern, "Discriminative learning of beam-search heuristics for planning," *International Joint Conference on Artificial Intelligence*, 2007.

[13] A. Y. Ng and S. J. Russell, "Algorithms for inverse reinforcement learning." 2000.

[14] P. Abbeel and A. Y. Ng, "Apprenticeship learning via inverse reinforcement learning," *International Conference on Machine Learning*, 2004.

[15] B. D. Ziebart, A. Maas, J. A. Bagnell, and A. K. Dey, "Maximum entropy inverse reinforcement learning," *National Conference on Artificial Intelligence*, 2008.

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

[17] S. Ross, G. J. Gordon, and J. A. Bagnell, "No-regret reductions for imitation learning and structured prediction," *Computing Research Repository*, vol. abs/1011.0686, 2010.

[18] 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.

[19] J. Schulman, J. Ho, A. Lee, I. Awwal, H. Bradlow, and P. Abbeel, "Finding locally optimal, collision-free trajectories with sequential convex optimization." *Robotics: Science and Systems*, 2013.

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