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 complex goals 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 geometric reasoning to address this challenge. We present an algorithm that searches the space of possible task and motion plans, and uses 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 (potential) 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 prior work.**

I. Introduction

We are interested in designing autonomous systems to 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 in TAMP is difficult; recent research has proposed several approaches [1], [2], [3], [4], [5], [6]. In this paper, we adopt the abstraction framework developed by Srivastava et al. [1] (henceforth referred to as SFRCRA-14) to factor the reasoning and search problems into interacting logical and geometric components.

In this work, we develop a complete algorithm for TAMP and propose learning methods to guide a joint 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 move, grasp, and putdown actions, while its refinement is a sequence of collision-free trajectories that implement the plan. We refer to the 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

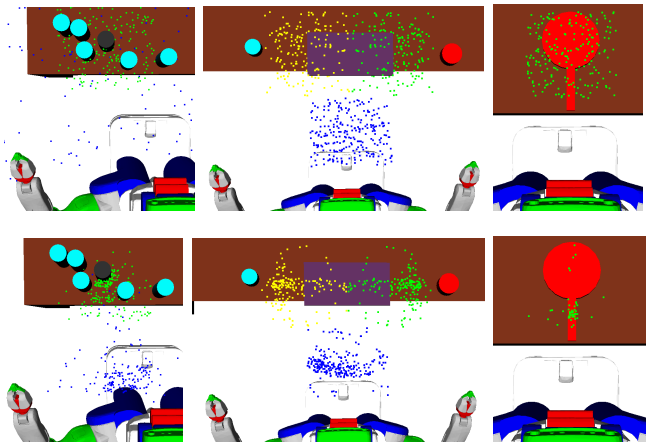

Fig. 1: We apply reinforcement learning to speed up planning for TAMP tasks. We break the problem down into a low-level policy that samples promising values for continuous parameters (e.g., pre-grasp poses, grasping poses, etc.), and a high-level policy that ranks different high-level plans. The above figures illustrate learning for the low-level system. The top images show initial, uniform distributions. The bottom images show learned distributions over base pose (blue) and end effector pose (green, yellow) for pick tasks in our simulated domains: a can (left), a tray (middle), and a frying pan (right). The green and yellow points refer to the positions of the tool center points; the end effectors are oriented to point toward the object being grasped. The can and frying pan are picked up using only one gripper.

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 an obstruction in a pick-place domain, the new plan may involve moving the obstruction out of the way. The key step in our approach uses this error information to define a *plan refinement graph*, where nodes are high-level plans and edges are unsatisfied preconditions that explain a failed attempt at refinement. We develop a complete algorithm that interleaves search over this graph with plan refinement.

Naturally, the space of high-level plans and their possible refinements is quite large. To combat this, we present machine learning techniques that guide search over both spaces.

At the low level, our system learns to propose continuous values for symbolic references that are likely to result in collision-free trajectories. Discretizing the space of continuous parameters is a common step in TAMP systems and largely relies on hand-coded heuristics or well-chosen sampling distributions.

We apply reinforcement learning (RL) to learn domain-specific distributions over these values in a domain-independent fashion. Our 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.

At the high level, we train heuristics that estimate how difficult it is to refine a given plan. Directly applying RL to this task is challenging because actions amount to either motion planning or task planning, and so are time consuming; there is also often a wide range of options among which to select. It is, however, fairly easy for a person to determine which high-level plans are promising, so we use inverse reinforcement learning based on expert demonstrations to train heuristics for this problem.

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 to learn probability models that map symbolic predicates to geometric states in TAMP problems. 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 an entire plan and its current refinement.

One set of approaches to TAMP involves search through a discretized space of high-level plans. Kaelbling et al. [3] search through a hierarchy of these spaces and use "geometric suggesters" to lazily perform this discretization. Dornhege et al. [5] define a way to integrate this discretization in classical planning. Garrett et al. [6] define a heuristic for TAMP that allows the search to incorporate geometric feasibility. Our approach could augment these systems by learning domain-specific discretizations of parameters.

An alternative set of approaches treats the high-level plan as defining a constraint satisfaction problem (CSP) where the parameters of the plan are the variables of the CSP. Lozano-Pérez and Kaelbling [8] discretize the parameter space and use a standard CSP solver. Lagriffoul et al. [4] and Garrett et al. [9] use a set of relaxed constraints to reduce the search space for a global solution. Toussaint [10] treats the problem as a global trajectory optimization. Our approach could be used to learn domain-specific initialization or discretization strategies for these approaches.

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(λ) [11] with function approximation to learn a value function for it. Their approach outperformed the previous state-of-the-art for this task and scaled to larger scheduling problems.

Zucker et al. [12] use RL to bias the distribution of a rapidly exploring random tree (RRT) for motion planning. They use features of a discretized workspace to train a non-uniform configuration space sampler with a policy gradient algorithm. In our work, we adapt their gradient updates to the TAMP framework.

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

- $\mathcal{T}$: a set of object *types* (e.g., movable objects, trajectories, poses, locations).
- $\mathcal{O}$: a set of *objects* (e.g., can_2, grasping_pose_6, location_3). $\mathcal{O}$ defines the configuration space of all movable objects, including the robot.
- $\mathcal{F}$: a set of *fluents*, which define relationships among objects and are Boolean functions defined over the configuration space.
- $\mathcal{I}$: a conjunction of fluents that defines the initial state.
- $\mathcal{G}$: a conjunction of fluents that defines the goal state.
- $\mathcal{U}$: a set of *high-level actions* (e.g., grasp, move, putdown), parameterized by objects and defined by *preconditions*, a set of fluents that describe when an action can be taken; and *effects*, a set of fluents 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 applied successively starting with $\mathcal{I}$, and the final state reached satisfies the goal condition $\mathcal{G}$.

B. Markov Decision Processes

Markov decision processes (MDPs) formalize the interaction between an agent and an environment. At each step of an MDP, the agent knows the current state and selects an action. This causes the state to change according to a known transition distribution.

Definition 2: 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, $\pi : \mathcal{S} \times \mathbb{Z} \rightarrow \mathcal{A}$, that maps the state and timestep to an action. The value function under π is a function of the timestep k and state s:

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

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. 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 [11].

In *inverse reinforcement learning* (IRL) [13], an agent attempts to recover R from a description of the MDP and execution traces of optimal behavior. This is useful in scenarios where an expert demonstrator can help guide learning. Some standard techniques include maximum-margin IRL [14] and maximum-entropy IRL [15].

IV. Solving Task and Motion Planning Problems

Solving TAMP problems requires evaluation of possible courses of action, each comprised of different combinations of instantiated action operators. A fundamental challenge is that the set of possible action instantiations 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 uses 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.

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.

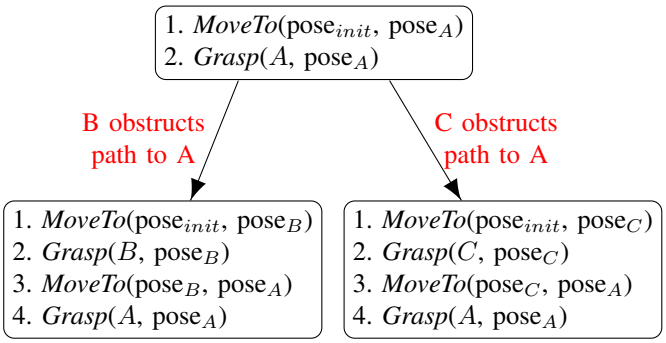

Fig. 2: A plan refinement graph for an environment with 3 objects: A, B, and C. The goal is to grasp A. Each node is a high-level plan, and edges are labeled with errors discovered during failed attempts at plan refinement. Each edge points to a plan that addresses that reason for failure. In this example, failures are obstructions and new plans move the offending object out of the way.

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 π_u and the current state of the search for a refinement. An edge (u, v) essentially represents a "correction" of π_u for a specific instantiation of the symbolic references in π_u. We 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 is an instantiation of references for a prefix $\pi_{u,k}$ of π_u, where 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 that incorporates the discovered facts $\varphi_{u,v}$ in the k^{th} state.

The overall search algorithm interleaves the search for feasible refinements of high-level plans with the addition of new edges and plan nodes into the PRG. 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 it defines. 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 a failed precondition for an infeasible refinement (e.g., collision-check the current set of trajectories to detect obstructions).

Different implementations of the non-deterministic choices in Alg. 1 can capture various search algorithms that account for 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 show that using trained search heuristics with the PRG can improve performance. 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. SFRCRA-14 uses a backtracking search over a discrete set of instantiations to implement REFINENODE. We want to learn policies for refinement, so we seek an algorithm that is more easily formulated 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.

```
   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.

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 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 feasibility, and returns a failed action index and associated predicate.

Randomized refinement has two key properties. The first

Algorithm *RandRef*(π, N_{max})

$\sigma \leftarrow$ INITREFINEMENT(π)
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.

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-place domain, it can make sense for the object's grasping 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 symbolic reference instantiation.

V. LEARNING HOW TO REFINE HIGH-LEVEL PLANS

A. Formulation as Markov Decision Process

We formulate plan refinement as the following MDP:

- States are tuples $\langle \pi, \sigma, E \rangle$ that consist of the high-level plan, its current (potentially infeasible) refinement, and the geometric environment.
- Actions are pairs $\langle p, x \rangle$, where p is the discrete symbolic reference to resample and x is the continuous value assigned to p in the new refinement.
- The transition distribution is defined by setting p's value to x. If x is IK feasible, the motion planner determines any corresponding trajectories.
- The reward function $R(s, a, s')$ is linearly interpolated between 0 and 20 based on the fraction of high-level actions whose preconditions are satisfied. Actions that result in an IK infeasible pose receive reward -1.
- The horizon is the number of available samples.
- $\mathcal{P}$ is a distribution over plans to be refined, defined as a distribution over planning problems.

Consider an example execution of refining a pick-place high-level plan. The robot must grasp an object and put it down at a certain location. The initial state consists of this plan, a set of initialized parameters, and meshes that describe the geometry of the problem. We'll suppose that the initial grasping pose causes both the grasp and the putdown actions to have violated preconditions. The first action in the MDP selects a new grasping pose; suppose it receives no reward because it does not work with the current putdown pose. The subsequent two actions then set consistent values for the grasping and putdown poses. This provides the maximum reward of 20 because all instantiated actions are now feasible.

B. Training Process

We restrict our attention to training policies that suggest x for a given parameter, since our refinement algorithm defines a way to select p. Our approach is to adapt the method of Zucker et al. [12], which uses a linear combination of features to define a distribution over poses. We learn a weight vector θ_p for each reference *type*, comprised of a pose type and possibly a gripper (e.g., "left gripper grasping pose," "right gripper putdown pose," "base pose").

We use 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 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, compute a high-level plan and run randomized refinement to attempt to find a valid plan refinement. Select actions according to the θ_p and collect rewards according to R. After every ϵ calls to RESAMPLE, take a gradient step on θ_p.

For a symbolic reference p, in state s, our policy selects a sample value x with probability

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

We define the expected reward of an episode, ξ, as $\eta(\theta_p) = \mathbb{E}_q[R(\xi)]$, 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. Then, for an appropriate step size α the weight vector update is:

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

We sample x from q using the Metropolis algorithm [16]. Since our distributions are continuous, calculating $\mathbb{E}_{q,s}[f]$ is hard, so we approximate it with a Monte Carlo estimate.

VI. LEARNING WHAT HIGH-LEVEL PLAN TO REFINE

A. Significance

In this section, we present a method for learning *which* high-level plan to try refining. Recall that in Alg. 1, the high level has a two-tiered decision to make: 1) which node in the PRG to visit next, and 2) whether to attempt to refine this node or generate failure information from it. These decisions are encoded in the routines NDGETNEXTNODE and NDCHOICE. Making these decisions intelligently is critical to good performance.

For example, consider a pick-place domain. Different sampled grasping poses may cause different obstruction errors to be propagated back up to the task planner. Figure 3 illustrates

two different failures, one of which generates a large number of obstructions. The PRG contains plans to remedy both of these issues; however, one will take much longer to refine than the other. An effective heuristic would allow us to allocate more search effort for the promising plan. In this section, we describe a method to learn one such heuristic.

B. Approach

Direct application of RL is difficult for two reasons: 1) the space of potential high-level plans is very large and reward are sparse; 2) actions take a long time to 'execute' as they often require motion planning. These combine to make the learning problem quite challenging, so we use IRL with expert demonstrations to train heuristics.

The heuristics are trained to trade off the computation time from refining a plan with that of generating a plan correction and adding a new plan to the PRG. We use u to represent the selected node and m the mode to apply (either trying to refine the node or quickly generating failure information).

We encode geometric features of a current plan refinement in a feature vector $f(u)$. We train separate weights to estimate the utility of each mode. We do this by stacking two copies of the features and using an indicator to zero out the top or bottom half: $f((u,m)) = \begin{bmatrix} f(u) \\ f(u) \end{bmatrix}^\top \begin{bmatrix} 1-m \\ m \end{bmatrix}$.

We obtain human-demonstrated trajectories (sequences of actions $(u,m)^*$) that intelligently navigate the PRG. We solve the following maximum-margin optimization [17] to recover a set of weights so that the expert's demonstration is (approximately) optimal:

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

The i iterate over the demonstrated trajectories and the j iterate over possible actions. The ξ are slack variables that ensure that a solution always exists. C is a regularization parameter that controls overfitting. At test time, we follow the policy encoded by w, picking the highest-scoring action at each step.

A single round of this approach often fails. The graphs generated by a human expert tend to look very different from those generated by the learned policy. We thus use dataset aggregation [18], a general approach to this problem where the expert provides new demonstrations on a set of states generated by executing the learned policy.

VII. Experiments

A. Methodology

We evaluate our approach in three 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 PRG, 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 error, replan with the task planner (which creates a child node), and repeat.

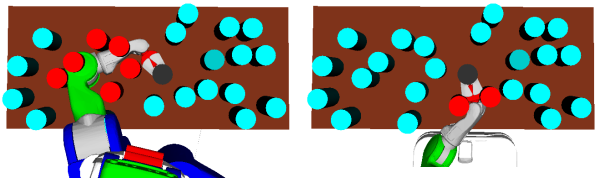

Fig. 3: In this problem, the goal is to grasp the black can. Obstructions make it so that this is not immediately possible; however, which obstructions are moved will have a big impact on how long it takes to find a solution (i.e., 6 obstructions in the left image vs. 2 obstructions in the right image). Each of these failures generates a new node in the plan refinement graph. We learn heuristics from expert demonstrations that allow the search to focus on the promising nodes in the graph.

We compare these baselines against two systems. The first combines learned refinement policies with the graph search used in baseline 2. The second is our full system, and uses learned refinement policies and graph search heuristics. For the dinner domain and frying domain, we focus only on the low-level learning. Since the errors propagated in these domains only relate to the stackability of objects, a good graph search strategy incorporates all available error information when attempting refinement. Thus, the graph search strategy from baseline 2 already performs well.

Initial experimentation revealed that jointly learning weights for all parameter types was intractable. Thus, we use a curriculum where the distribution of planning problems, $\mathcal{P}$, gets progressively harder. For the full system, we train the refinement policies first, then fix them while collecting demonstrations and training the graph search heuristics. To reduce variance in the process, we train 3 sets of refinement weights independently and select the one that performs best on a validation set.

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.

Many of our features use the concept of a *cone*, parameterized by an apex, an angle range, a direction, and a number of buckets. To learn grasping poses for a target object o, we use 25 features. 9 binary features bucket the distance between the sample and o. 9 binary features bucket the sample height. 3 features describe the number of objects within discs of radius 7, 10, and 15 centimeters around the sample. 3 binary features bucket the sample within a cone of apex o and angle range 0 to π, directed toward the robot. The final feature compares the 2-norm of the difference between the previous sample and the current one with 10^{-3}.

To learn putdown poses for a target location l, we use 24

# 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
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, standard deviation, and average refinement time & number of motion planning calls on successes, for baseline 1 (T); baseline 2 (B); learned refinement policies with greedy graph search (L); and our full system: learned refinement policies and graph search heuristics (F). L and F results are averaged across 5 separately trained sets of weights. Time limit: 300s.

features. The first 18 are the same as for grasping. 5 binary features bucket the sample within a cone of apex l and angle range 0 to π (with the last feature describing orientation), directed toward the closest table edge. The final feature is the same as for grasping.

To learn base poses for interacting with an object o, we use 18 features. The first 9 are the same as for grasping. 7 binary features bucket the sample within a cone of apex o and angle range 0 to π (with the last feature describing orientation), directed toward the closest table edge. 1 feature describes whether the sampled pose collides with unmovable objects in the environment. 1 feature describes whether there are obstructions in front of the sampled pose.

To learn putdown locations l, we use 5 features. 1 binary feature encodes whether any objects are within 15 centimeters of l. 1 feature is the distance from l to the closest table edge. 3 features count the number of objects in each bucket of a cone with apex l and angle range 0 to $3\pi/4$, directed toward the closest table edge.

Our high-level heuristics use features to describe the feasibility of plans. In our domains, this mainly summarizes issues with potential grasps. We use three features to evaluate a potential grasp action, targeted at an object o. We consider

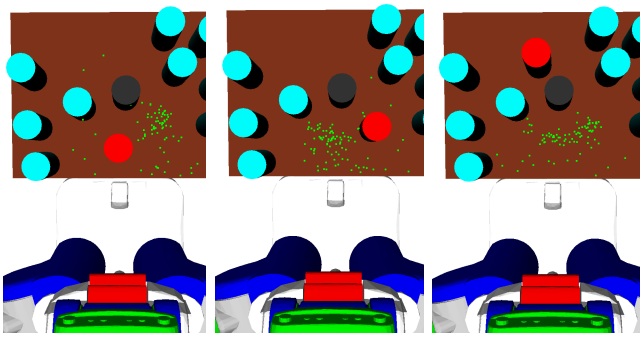

Fig. 4: The learned left arm grasping (green) distribution for the black can adapts as a single potential obstruction (red) is moved.

a cone whose point is at the object center and ranges from angles $-\frac{\pi}{3}$ to $\frac{\pi}{3}$, pointing toward the closest table edge from o. The exists_obstr feature is a binary variable that indicates whether any other objects lie in this cone. The exists_path feature is a binary variable that indicates whether there is a linear grasp path wide enough for the robot's gripper to fit through within the cone. The sweep_count feature computes the minimum number of collisions with the robot's gripper placed at any angle inside the cone.

We construct these features for the first five grasp actions in the plan, padding with -1 as necessary. 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 u was picked for refinement, and 4) the number of times u was picked for generating an error.

Our experiments are conducted in Python 2.7 using the OpenRave simulator [19] with a PR2 robot. The motion planner we use is trajopt [20], and the task planner is Fast-Forward [21]. The experiments were carried out in series on an Intel Core i7-4770K machine with 16GB RAM. The time limit for each problem was 300 seconds. Table I summarizes our quantitative results.

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 disable the right gripper, so any obstructions to the target object must be picked up and placed elsewhere on the table. A typical high-level plan in this domain has around 6 steps. This domain has 4 types of continuous references: base poses, object grasping poses, object putdown poses, and object putdown locations.

Our curriculum learning system trains distributions for base poses and grasping poses for 12 iterations with $\epsilon = 5$, then base poses, grasping poses, and putdown poses (at fixed location) for 18 iterations with $\epsilon = 20$, then all reference types for 30 iterations with $\epsilon = 20$. We fix $H = 100$.

To train the graph search heuristics, we collected approximately 300 optimal actions from the human demonstrator, over 3 rounds of dataset aggregation. After these 3 rounds, performance plateaued. We use $C = 10^9$ to solve the maximum-margin optimization problem.

The results demonstrate significant improvements in performance when compared to the baseline systems for success rate. When backtracking search succeeds, it does so quickly, so it has a faster average refinement time on the successes. Figure 1 shows learned base motion and pickup distributions. Figure 4 shows how the learned distribution shifts based on the geometric context.

C. Dinner Domain

We run one set of experiments, using 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. A typical high-level plan in this domain has around 33 steps. This domain has 5 types of continuous references: base poses, object grasping 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 20 iterations, then object grasping and putdown poses for 20 iterations. We fix $H = 100$ 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 1 shows learned tray pickup poses after all 20 iterations.

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. A typical high-level plan in this domain has around 45 steps. This domain has 3 types of continuous references: base poses, pan grasping poses, and pan putdown poses.

We did not need curriculum learning. We fix $N = 30$, $H = 100$, and $\epsilon = 5$. SFRCRA-14 did not have a frying domain, so we use the following hand-coded distribution for picking up the pans: 4 grasping poses in the cardinal directions around the lip of the pan, and 4 equidistant along the handle.

The results demonstrate significantly higher success rate versus the baseline systems. Similar to the can domain, when backtracking "gets lucky" and picks grasping poses along the handle early in the search, it succeeds quickly, leading to a faster average refinement time on the successes. Figure 1 shows learned frying pan grasping poses after all 30 iterations. Our system learned to prefer picking up the pan at its handle to fit it into the shelf, which is not shown.

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.

Acknowledgments

This research was funded in part by the Intel Science and Technology Center (ISTC) on Robotics and Embedded Systems. Dylan is also supported by an NSF GRFP fellowship and a Berkeley fellowship. Siddharth is supported by United Technologies Research Center; all opinions in this work are of the authors, not the organization.

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. 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.

[6] 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.

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

[8] T. Lozano-Pérez and L. P. Kaelbling, "A constraint-based method for solving sequential manipulation planning problems," in *IEEE Conference on Intelligent Robots and Systems*, 2014.

[9] C. R. Garrett, T. Lozano-Perez, and L. P. Kaelbling, "Backward-forward search for manipulation planning," *IEEE Conference on Intelligent Robots and Systems*, 2015.

[10] M. Toussaint, "Logic-geometric programming: An optimization-based approach to combined task and motion planning," in *International Joint Conference on Artificial Intelligence*, 2015.

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

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

[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, pp. 327–335, 1995.

[17] B. Taskar, V. Chatalbashev, D. Koller, and C. Guestrin, "Learning structured prediction models: A large margin approach," in *Proceedings of the 22nd international conference on Machine learning*. ACM, 2005, pp. 896–903.

[18] S. Ross, G. J. Gordon, and J. A. Bagnell, "No-regret reductions for imitation learning and structured prediction," *Computing Research Repository*, 2010.
[19] 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.
[20] 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.
[21] Jörg Hoffman, "FF: The fast-forward planning system," *AI Magazine*, vol. 22, pp. 57–62, 2001.