Learning Plan Refinement Graph Exploration in Combined Task and Motion Planning

Edward Groshev[1] and Christopher Lin[1]

***Abstract*— In mobile manipulation planning, it is not uncommon for tasks to require thousands of individual motions. Such problems require reasoning about courses of action from the viewpoint of logical objectives as well as the feasibility of individual movements in the configuration space. In discrete representations, planning complexity is exponential in the length of the plan; in mobile manipulation, the set of parameters for an action is often continuous, so we must also cope with an infinite branching factor. *Task and motion planning* (TAMP) methods integrate logical search with continuous geometric reasoning to address this challenge. Our recent work in TAMP has developed a *plan refinement graph*, a data structure which holds candidate task plans that address different infeasibilities. The work included a preliminary technique for learning to explore this graph. In this paper, we improve this approach, developing techniques for using statistical machine learning and learning from demonstrations to guide the search process. Our methods learn from human-demonstrated optimal trajectories through the space of available task plans. Our contributions are as follows: 1) we formulate navigation through a plan refinement graph as an MDP; 2) we present a method that trains heuristics for intelligently searching the available space of task plans, through learning from expert demonstrations; and 3) we run experiments to evaluate the performance of our system. We show improvements in performance over the systems we build on.**

I. Introduction

A long-term goal of robotics research is the introduction of intelligent household robots. To be effective, such robots will need to perform complex tasks over long horizons (e.g., setting a dinner table, doing laundry). Planning for these long-horizon tasks is infeasible for state-of-the-art motion planners, making the need for a hierarchical system of reasoning apparent.

One way to approach hierarchical planning is through combined *task and motion planning* (TAMP). In this approach, an agent is given a symbolic, logical characterization of actions (e.g., move, grasp, putdown), along with a geometric encoding of the environment. TAMP systems maintain a hierarchical separation of high-level, symbolic task planning and low-level, geometric motion planning. Efficient integration of these two types of reasoning is difficult, and recent research has proposed several methods for it [1], [2], [3], [4], [5]. We adopt the principles of abstraction in the TAMP system developed by Srivastava et al. [1] (henceforth referred to as SFRCRA-14) to factor the reasoning and search problems into interacting logic-based and geometric components.

Recent work [6] that builds on SFRCRA-14 proposes methods for jointly carrying out guided search in the space of high-level (logic-based) plans and their low-level *refinements*, instantiations of continuous values for symbolic references in the plan. We refer to this search for a valid low-level refinement as *plan refinement*. In this paper, we improve upon the recent work's preliminary methods for high-level search, while retaining its methods for learning low-level refinement distributions.

[1]{ronuchit, eddiegroshev}@berkeley.edu

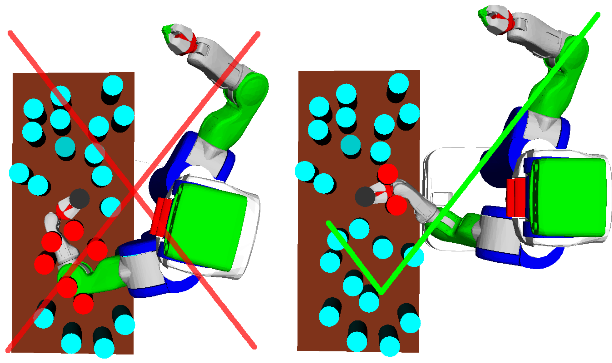
Fig. 1: When trying to grasp the black can, the grasping pose sampled by a robot can significantly affect the quality of the obstructions determined. In the left image, 6 objects (shown in red) obstruct the grasping trajectory, but in the right one, only 2 do. The new plan proposed from the right image thus only requires moving 2 obstructions out of the way. In this work, we integrate machine learning into our task and motion planning system using expert demonstrations, to learn an ordering on plan exploration. The system learns to prefer simpler and more feasible plans, such as the one resulting from the scenario on the right.

As in Chitnis et al. [6], we make use of unpublished work recently submitted for review to ICRA 2016. The work develops a *plan refinement graph*, a data structure that stores a set of qualitatively different high-level plans that could solve a task (the graph is described in detail in Section IV-B). During planning, we repeatedly select one such plan and either 1) try to refine it or 2) incorporate geometric information in order to generate a new high-level plan. This allows interleaving plan refinement with a search over *which* high-level plan to try refining, given options that address different infeasibilities.

We present machine learning techniques used in learning from demonstrations to train heuristic functions that guide the search process over the high level. This amounts to learning to predict how difficult it is to refine a given plan. Many TAMP systems rely on hand-coded heuristics to achieve this. We develop a system that uses human demonstrations of optimal plan-space trajectories to learn intelligent navigation of the plan refinement graph. We use 1) a max-margin formulation, which is commonly used in inverse reinforcement learning, along with Dataset Aggregation (DAgger) [7] and 2) a discriminative formulation using logistic regression.

The contributions of our work are as follows: 1) we formulate navigation through a plan refinement graph as an MDP; 2) we present a method that trains heuristics for intelligently searching the available space of task plans, through learning from expert demonstrations; and 3) we run experiments to evaluate the performance of our system in a variety of simulated domains. We show improvements in performance over both SFRCRA-14 and the preliminary system developed by Chitnis et al. [6].

II. Related Work

Our work uses machine learning techniques common in learning from demonstrations to improve planning reliability in a TAMP system.

Kaelbling et al. [2] use hand-coded "geometric suggesters" to propose continuous geometric values for the plan parameters. These suggesters are heuristic computations which map information about the robot type and geometric operators to a restricted set of values to sample for each plan parameter.

Lagriffoul et al. [3] propose a set of geometric constraints involving the kinematics and sizes of the specific objects of interest in the environment. These constraints then define a feasible region from which to search for geometric instantiations of plan parameters.

Garrett et al. [4] use information about reachability in the robot configuration space and symbolic state space to construct a *relaxed plan graph* that guides motion planning queries, using geometric biases to break ties among states with the same heuristic value.

By contrast to these methods, which can be viewed as hand-coding methods for refining a single plan, we learn methods for searching the space of high-level task plans. This constitutes a meta-level search, which can be augmented by any of the described approaches, or by the approach for learning refinement distributions used in Chitnis et al. [6] (as in this paper). To our knowledge, our work is the first to use learning from demonstrations to guide search over the space of task plans in a TAMP system.

Another line of work has been devoted to using machine learning techniques for training heuristics to guide search algorithms. This general formulation has been applied to many domains other than hierarchical planning for robotics.

Boyan et al. [8] present an application to solving optimization problems using local search routines. They describe an algorithm, STAGE, for learning a state evaluation function using features of the optimization problem. This function then guides the local search toward better optima.

Arbelaez et al. [9] use machine learning to solve constraint satisfaction problems (CSPs). Their approach, Continuous Search, maintains a heuristic model for solving CSPs and alternates between two modes. In the *exploitation* mode, the current heuristic model is used to solve user-inputted CSPs. In the *exploration* mode, this model is refined using supervised learning with a linear SVM.

Xu et al. [10] apply machine learning for classical task planning, drawing inspiration from recent advances in discriminative learning for structured output classification. Their system trains heuristics for controlling forward state-space beam search in task planners.

III. Background

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

A. Task and Motion Planning

A motion planning problem is defined as a tuple $\langle \mathcal{X}, f, p_0, p_t \rangle$, where $\mathcal{X}$ is the space of possible configurations or poses of a robot, f is a Boolean function that determines whether or not a pose is in collision, and $p_0, p_t \in C$ are the initial and final poses. The solution to a motion planning problem is a collision-free trajectory that connects p_0 and p_t. To allow for object manipulation, we let $\mathcal{X}$ include poses for each movable object in the environment.

In task and motion planning, we add more abstract concepts to this formulation, including *fluents* (logical properties that hold either true or false and may vary across configurations) and *actions* (operations that the agent may choose to execute, resulting in changes to the configuration and the set of fluents that are true). Each action may require motion planning prior to execution. The overall problem is to plan the sequence of actions that the agent can execute to achieve a desired goal condition expressed in terms of fluents.

For example, we can use the action schema *grasp(Object o, Manipulator p, GraspingPose g, Trajectory m)* to abstractly represent grasping an object, o. In order to apply this action, the agent must select values for each of these parameters (e.g., *grasp(can*$_1$*, left,* g_1*,* m_1*)*). These *instantiated* actions change the value of specific fluent instantiations (also called fluent *literals*), such as *in-gripper(can*$_1$*, gripper*$_l$*)* and *empty(gripper*$_l$*)*.

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

- $\mathcal{O}$ is a set of objects denoting elements such as cans, trajectories, and poses. Note that $\mathcal{O}$ includes the configuration space of all movable objects, including the robot.
- $\mathcal{T}$ is a set of object *types*, such as movable objects, motion plans, poses, and locations.
- $\mathcal{F}$ is a set of *fluents*, which define relationships among objects and are Boolean functions defined over the configuration space.
- $\mathcal{I}$ is the set of fluent literals that hold true in the initial state.
- $\mathcal{G}$ is the set of fluent literals defining the goal condition.
- $\mathcal{U}$ is a set of *high-level actions* that are parameterized using objects and defined by *preconditions*, a set of fluent literals that must hold true in the current state to be able to perform the action; and *effects*, a set of fluent literals that hold true after the action is performed.

An instantiated action is said to be *feasible* in a state if and only if its preconditions hold in that state. Implicitly, the trajectories corresponding to actions must be collision-free.

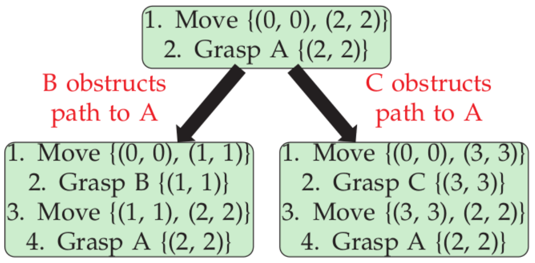

Fig. 2: A simple plan refinement graph for an environment with 3 objects: A, B, and C. The goal is to grasp object A. Each node maintains a high-level plan and a set of instantiated continuous values for its symbolic references. The edges are labeled with errors discovered by the low-level motion planner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

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 an MDP as a tuple $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma, \mathcal{P} \rangle$, where

- $\mathcal{S}$ is the state space.
- $\mathcal{A}$ is the action space.
- $T(s, a, s') = Pr(s'|s, a)$ for $s, s' \in \mathcal{S}, a \in \mathcal{A}$ is the transition distribution.
- $R(s, a, s')$ for $s, s' \in \mathcal{S}, a \in \mathcal{A}$ is the reward function.
- $\gamma \in [0, 1]$ is the discount factor.
- $\mathcal{P}$ is the initial state distribution.

A solution to an MDP is a policy, $\pi : \mathcal{S} \rightarrow \mathcal{A}$, that maps states to actions. The value, $V_\pi(s)$, of a state under π is the sum of expected discounted future rewards generated from starting in state s and selecting actions according to π:

$$V_\pi(s) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t R(s_t) | \pi, s_0 = s].$$

The optimal policy, π^*, maximizes this value for all states.

IV. Solving Task and Motion Planning Problems

A. SFRCRA-14

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.

SFRCRA-14 solves TAMP problems by: incrementally searching for a high-level plan that solves the logical abstraction of the given TAMP problem; determining a prefix of the plan that has a motion planning feasible refinement; updating the high-level abstraction to reflect the reason for infeasibility; and searching for a new plan suffix from the failure step onwards. This search process addresses the fundamental TAMP problem: high-level logical descriptions are lossy abstractions of the true environment dynamics and thus may not include sufficient information to determine the true applicability of a sequence of actions.

In general, including geometric properties in the logic-based formulation leads to an increase in the number of objects representing distinct poses and/or trajectories. For instance, expressing the fact that a trajectory for grasping can_1 is obstructed by can_3 from the current pose of the robot would require setting a fluent of the form *obstructs(*can_3*,* $pose_{17877}$*,* $trajectory_{3219}$*,* can_1*)* to true in the description of the high-level state. In turn, this would require adding $pose_{17877}$ and $trajectory_{3219}$ into the set of objects if they were not already included. Unfortunately, the size of the abstracted, logic-based state space grows exponentially with the number of objects, and such an approach quickly leads to unsolvable task planning problems.

SFRCRA-14 addresses this challenge by abstracting the continuous action arguments, such as robot grasping poses and trajectories, into a *bounded* set of symbolic references to potential values. A *high-level*, or *symbolic*, plan refers to the fixed task sequence returned by a task planner and comprised of these symbolic references. An *interface layer* conducts plan refinement, searching for instantiations of continuous values for symbolic references while ensuring action feasibility. The resulting process is able to utilize off-the-shelf task and motion planners while carrying out the necessary exchange of information in a scalable manner.

B. Plan Refinement Graph

Unpublished work recently submitted for review to ICRA 2016 builds on SFRCRA-14 and develops a complete algorithm that maintains a *plan refinement graph* (PRGraph). Every node u in the PRGraph represents a high-level plan π_u and the current state of the search for a refinement. An edge (u, v) in the PRGraph represents a "correction" of π_u for a specific instantiation of the symbolic references in π_u. Let $\pi_{u,k}$ be the plan prefix of π_u consisting of the first k actions. Formally, each edge $e = (u, v)$ is labeled with a tuple $\langle \sigma, k, \varphi \rangle$. σ denotes an instantiation of references for a prefix $\pi_{u,k}$ of π_u such that feasible motion plans have been found for all previous actions $\pi_{u,k-1}$. φ denotes a conjunctive formula consisting of fluent literals that were required in the preconditions of the k^{th} action in π_u but were not true in the state obtained upon application of $\pi_{u,k-1}$ with the instantiation σ_k. The plan in node v (if any) retains the prefix $\pi_{u,k-1}$ and solves the new high-level problem which incorporates the discovered facts $\varphi_{u,v}$ in the k^{th} state.

The overall search algorithm then interleaves the search for feasible refinements of each high-level plan with the addition into the PRGraph of new edges and plan nodes using the semantics described above.

Figure 2 illustrates a simple example of a PRGraph.

C. Previous Approach for Learning to Search the PRGraph

Chitnis et al. [6] explain that the high level has a two-tiered decision to make: which node in the PRGraph to visit next, and whether to 1) attempt to refine this node or 2) quickly generate failure information from it, thus spawning a child node. They develop heuristics that are trained to estimate the difficulty associated with refining a plan, in order to make this decision intelligently.

To select among the potential refinement options, the authors train two decision tree regressors that determine how many iterations would be needed to achieve a valid refinement for a node, or for a child of that node which incorporates a single additional geometric fact about the environment. To obtain an estimate for a full plan, they sum this number across all of the plan's actions. The implicit assumption being made here is that dependencies in plans are, in some way, local: the plan must be factorizable into subportions with independent refinements.

At test time, they use the regressors to find the best two-tiered action to take at each step. For example, if refining a child node would reduce the number of steps to a valid refinement, they bias toward generating a child node.

V. Learning to Search the PRGraph

A. Formulation as MDP

We adopt the same two-tiered decision strategy described in Section IV-C. We formulate PRGraph search as an MDP as follows:

- A state $s \in \mathcal{S}$ is a tuple (PRG, r_{cur}, E), consisting of the PRGraph with all its high-level plans, the current setting of values for symbolic references for all high-level plans, and an encoding of the geometric environment.
- An action $a \in \mathcal{A}$ is a pair (n, m), where n is the selected node and m is the mode to apply (either trying to refine the node or quickly generating failure information).
- If we select a node n to try to refine, the transition function $T(s, a, s')$ is the same as that in the plan refinement MDP presented in Chitnis et al. [6] Briefly, it describes how continuous values for symbolic references in node n's plan are resampled, altering r_{cur}. If we instead try to quickly generate failure information for n, then T is defined by how we determine a geometric fact to replan with at the task planner level. In our system, we sample a trajectory (which may have collisions) for achieving each step in the plan, then roll out that trajectory in simulation and propagate back to the task planner the first error we encounter (obstructing object, unreachable object, etc.).
- The reward function $R(s, a, s')$ is 1 if r_{cur} for any high-level plan in the PRGraph has collision-free linking trajectories, and 0 otherwise.

B. Max-Margin Approach

We exploit properties of the environment to hand-design a feature vector $f(n)$ that geometrically describes aspects of a single high-level plan, thus encoding its refinability. 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 action that can be taken from a state, we can obtain human-demonstrated trajectories (sequences of actions $(n, m)^*$) that intelligently navigate the PRGraph. We then solve the following max-margin optimization problem with a structured margin and slack variables:

$$\min ||w||^2 + C \sum_i \xi_i$$

$$\text{s.t. } w^\top f((n,m)_i^*) \geq w^\top f((n,m)_{ij}) +$$

$$d(f((n,m)_i^*), f((n,m)_{ij})) - \xi_i \ \forall i, j, \ \ \xi_i \geq 0 \ \forall i$$

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. We note that the weight vector w encodes a score function on the different actions, which produces an ordering that matches the Q-function in the described MDP. Of course, the score function is not identical to the Q-function, because we are not incorporating $R(s, a, s')$ in our formulation for w.

We use DAgger to augment our training data. In the first iteration, we navigate the PRGraph following the expert actions and store these actions in a dataset. After some iterations, we use the dataset to solve for an initial weight vector w. On subsequent iterations, we navigate the PRGraph by rolling out the policy encoded by w (pick the action with the highest score $w^\top f((n, m))$) while asking the expert what the optimal action would be at each step, appending these into the dataset. After some iterations, we use the aggregated dataset to solve for a new w, and repeat.

At test time, we follow the policy encoded by w straightforwardly, picking the highest-scoring action at each step.

C. Logistic Regression Approach

Within the framework of classification, a policy can be viewed as a classifier that discriminates between "good" and "bad" actions. While it is natural to model the quantity $p(action|state)$, which can be viewed as a multi-class classification problem, the action space support varies with the number of nodes added to the PRGraph. Instead, we relax the problem and model the quantity $p(optimal|action, state)$. This is simply a binary classification problem where the label takes on two possible values: "optimal" and "not optimal". Using a logistic regression model we express the conditional probability as

$$p(optimal = true|action, state; w) = \frac{1}{1 + e^{-w^\top x}}$$

where $x = f((n, m))$ is the feature vector associated with the current action. To estimate the parameter vector w we

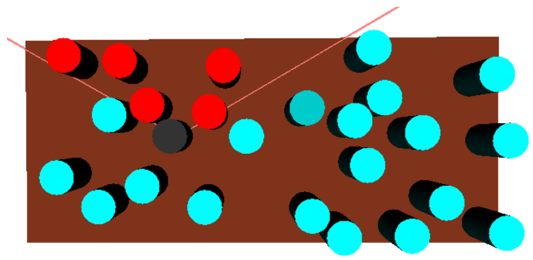

Fig. 3: If the target object to be grasped is the black can, we consider the objects whose centers lie in a cone from angles $-\frac{\pi}{3}$ to $\frac{\pi}{3}$ toward the closest table edge in our feature computations. These objects are shown here in red.

minimize the following cost function:

$$\ell(w|D) = \lambda||w||_2^2 - \sum_{ij} y_{ij} \log \mu_{ij} + (1-y_{ij})\log(1-\mu_{ij})$$

where $\mu_{ij} = 1/(1+e^{-w^\top f((n,m)_{ij})})$ and $y_{ij} = 1$ if the human took action j in the i'th demonstration and 0 otherwise. We make the assumption that humans performs optimally.

For this approach we do not explicitly use DAgger to augment the training data, instead we train on the largest data set that was collected. At test time, we evaluate a stochastic policy and a deterministic policy. The deterministic policy takes the action with the highest probability of being optimal. For the stochastic policy we normalize the probability of being optimal across all possible actions and randomly select an action from the resulting distribution.

VI. Experiments

We evaluate our approach in the can domain, which has cans uniformly distributed on a table. We compare performance with two baselines. Baseline 1 is SFRCRA-14, which uses the following fixed graph search policy: try 3 times to refine the deepest node in the PRGraph; if unsuccessful, generate a geometric fact from it, replan (which creates a child node), and repeat. The number 3 was chosen to trade off between providing each node a fair opportunity to be refined (which could fail a couple of times if sampling produces undesirable values), and not spending too much time before giving up and generating a child node to try next. Baseline 2 is the prior work by Chitnis et al. [6], described in Section IV-C.

We run four sets of experiments, using 25, 30, 35, and 40 cans on the table. The goal across all experiments is for the robot to pick up a particular object with its left gripper. We disabled the right gripper, so any obstructions to the target object must be picked up and placed elsewhere on the table. This domain has 4 types of continuous references: base poses, object grasp poses, object putdown poses, and object putdown locations onto the table.

We now describe the 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

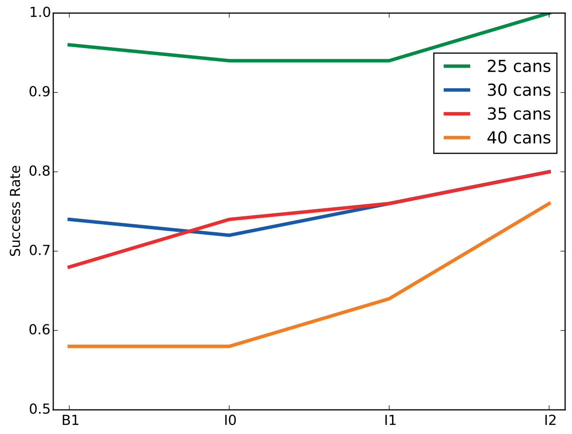

Fig. 4: Line graph illustrating success rates on the test set for baseline 1 and our system after each round of DAgger, using the abbreviations described in the table caption. Our method performs much better than the baseline after 2 iterations of DAgger. We also note the upward trend in success rate across the iterations.

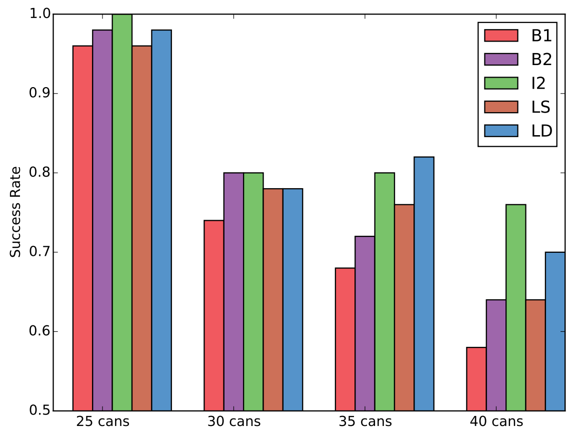

Fig. 5: Bar graph illustrating success rates on the test set for the two baselines, the two logistic regression policies, and our max-margin formulation after 2 iterations of DAgger. The max-margin approach achieved much higher success rates than both baselines, in general. The deterministic policy trained with logistic regression had comparable results.

$-\frac{\pi}{3}$ to $\frac{\pi}{3}$ toward the closest table edge from o, as shown in Figure 3. 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

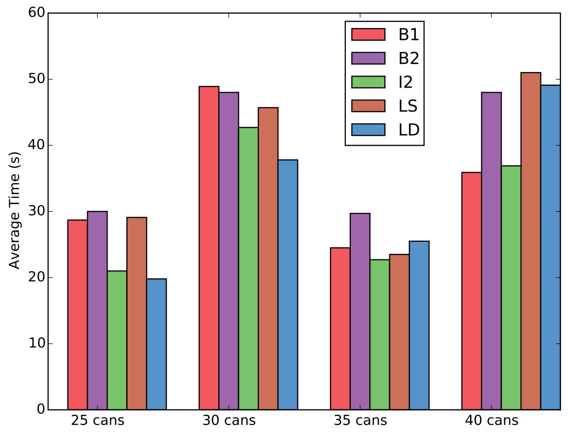

Fig. 6: Bar graph illustrating average total times on the test set for the two baselines, the two logistic regression policies, and our max-margin formulation after 2 iterations of DAgger. The max-margin approach produced noticeably faster performance than both baselines, in general. The deterministic policy trained with logistic regression had comparable results. In the 40 can environment it had an increased planning time compared to the baseline, but this is OK because in more challenging environments the baseline can only solve simple instantiations which naturally have a lower planning time.

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) a counter for how many times n was picked for refinement, 4) a counter for how many times n was picked for generating an error, 5) the max of 0 and the difference between expected plan length and the current plan length, this indicates how much shorter the current plan is, and 6) the analog of the previous feature to indicate how much longer than expected the current plan is. The expected plan length is approximated as a linear function of sweep_count. For the logistic regression approach we added a bias term to the feature vector.

We report results on fixed test sets of 50 randomly generated environments per number of objects. At each iteration of training, we collect approximately 100 optimal actions from the human demonstrator (using DAgger, this means collect 100 demonstrations, construct w, collect 100 more while rolling out w, construct new w, repeat). We found that after 2 rounds of DAgger, performance plateaued, so we stopped there.

Our experiments are conducted in Python 2.7 using the OpenRave simulator [11] with a PR2 robot. The motion planner we use is trajopt [12], and the task planner is Fast-Forward [13]. The experiments were carried out in series on an Intel Core i7-4770K machine with 16GB RAM. For the max-margin optimization, we used a constant margin of $d = 1$, and we set $C = 10^9$, indicating that we want to penalize violating constraints significantly and are willing to have $||w||$ grow. In our experiments, though, $||w||$ did not actually become so large that overfitting was an issue. We used Gurobi for solving the quadratic program.

Table I summarizes our quantitative results. Figure 1 illustrates a scenario where our learning methods prove important. Figure 4, Figure 5, and Figure 6 show our quantitative results in graphs. The results demonstrate significant improvements in performance to the baseline systems for success rate and average overall time. The number of calls to the motion planner remained roughly the same, suggesting that the performance benefits instead derive from less time wasted on inverse kinematics checks when attempting to refine plans with no valid refinement.

VII. Conclusion and Future Work

In this work, we developed methods for guiding search through a plan refinement graph in a TAMP system, using machine learning based on expert demonstrations. Thus, the system learned to intelligently navigate this graph, making a decision of both which node to explore next and whether to attempt to refine this node or generate a geometric fact for replanning. We evaluated performance against a baseline strategy and a previous approach to the problem, and we found that our method outperformed both substantially. Future work includes trying different inverse reinforcement learning algorithms, using non-linear combinations of the features (e.g. by learning a neural network to score actions), and expanding our approach to different experimental domains.

VIII. Acknowledgements

We thank Rohan Chitnis and Dylan Hadfield-Menell for their collaboration, advice, and feedback throughout this project.

References

[1] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel, “Combined task and motion planning through an extensible planner-independent interface layer,” *IEEE Conference on Robotics and Automation*, 2014.

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

[3] F. Lagriffoul, D. Dimitrov, J. Bidot, A. Saffiotti, and L. Karlsson, “Efficiently combining task and motion planning using geometric constraints,” 2014.

[4] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, “FFRob: An efficient heuristic for task and motion planning,” in *International Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2014. [Online]. Available: http://lis.csail.mit.edu/pubs/garrett-wafr14.pdf

[5] C. Dornhege, P. Eyerich, T. Keller, S. Trüg, M. Brenner, and B. Nebel, “Semantic attachments for domain-independent planning systems,” in *Towards Service Robots for Everyday Environments*. Springer, 2012, pp. 99–115.

[6] R. Chitnis, D. Hadfield-Menell, S. Srivastava, A. Gupta, and P. Abbeel, “Learning an interface to improve efficiency in combined task and motion planning,” in *IROS Workshop on Machine Learning in Planning and Control of Robot Motion (MLPC)*, 2015.

[7] S. Ross, G. J. Gordon, and J. A. Bagnell, “A reduction of imitation learning and structured prediction to no-regret online learning,” *arXiv preprint arXiv:1011.0686*, 2010.

[8] J. Boyan and A. W. Moore, “Learning evaluation functions to improve optimization by local search,” *J. Mach. Learn. Res.*, vol. 1, pp. 77–112, Sep. 2001. [Online]. Available: http://dx.doi.org/10.1162/15324430152733124

[9] A. Arbelaez, Y. Hamadi, and M. Sebag, “Continuous search in constraint programming,” in *Autonomous Search*, Y. Hamadi, E. Monfroy, and F. Saubion, Eds. Springer Berlin Heidelberg, 2012, pp. 219–243. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21434-9_9

[10] Y. Xu, S. Yoon, and A. Fern, “Discriminative learning of beam-search heuristics for planning,” in *Proceedings of the International Joint Conference on Artificial Intelligence*, 2007, pp. 2041–2046.

[11] R. Diankov and J. Kuffner, “Openrave: A planning architecture for autonomous robotics,” Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-08-34, July 2008.

[12] J. Schulman, J. Ho, A. Lee, I. Awwal, H. Bradlow, and P. Abbeel, “Finding locally optimal, collision-free trajectories with sequential convex optimization.” in *Robotics: Science and Systems*, vol. 9, no. 1. Citeseer, 2013, pp. 1–10.

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

# Cans	System	% Solved	Avg Time (s)	Avg # MP Calls
25	B1	96	28.7	22
25	B2	98	30.0	21
25	I0	94	24.8	21
25	I1	94	28.5	19
25	I2	100	21.0	18
25	LS	96	29.1	23
25	LD	98	19.8	18
30	B1	74	48.9	29
30	B2	80	48.0	29
30	I0	72	40.0	29
30	I1	76	44.2	28
30	I2	80	42.7	30
30	LS	78	45.7	35
30	LD	78	37.8	29
35	B1	68	24.5	15
35	B2	72	29.7	15
35	I0	74	31.1	15
35	I1	76	22.4	14
35	I2	80	22.7	14
35	LS	76	23.5	15
35	LD	82	25.5	15
40	B1	58	35.9	15
40	B2	64	48.0	19
40	I0	58	54.5	22
40	I1	64	36.6	14
40	I2	76	36.9	16
40	LS	64	51	22
40	LD	70	49.1	18

TABLE I: Percent solved, average overall time, and average number of calls to the motion planner for baseline 1 (B1), baseline 2 (B2), logistic regression with stochastic policy (LS), logistic regression with deterministic policy (LD), and max-margin with DAgger after iteration 0 (I0), 1 (I1), and 2 (I2). Iteration 0 evaluates performance before DAgger was run (so just the initial set of 100 demonstrations), and iterations 1 and 2 are after one and two rounds of DAgger, respectively. Average time and number of motion planner calls were computed across the subset of environments for which all systems succeeded, which explains why they do not scale with the number of cans. Time limit: 300s.