Learning Sequential Decision Tasks for Robot Manipulation with Abstract Markov Decision Processes and Demonstration-Guided Exploration

David Kent*, Siddhartha Banerjee*, and Sonia Chernova*

***Abstract*— Solving high-level sequential decision tasks situated on physical robots is a challenging problem. Reinforcement learning, the standard paradigm for solving sequential decision problems, allows robots to learn directly from experience, but is ill-equipped to deal with issues of scalability and uncertainty introduced by real-world tasks. We reformulate the problem representation to better apply to robot manipulation using the relations of Object-Oriented MDPs (OO-MDPs) and the hierarchical structure provided by Abstract MDPs (AMDPs). We present a relation-based AMDP formulation for solving tabletop organizational packing tasks, as well as a demonstration-guided exploration algorithm for learning AMDP transition functions inspired by state- and action-centric learning from demonstration approaches. We evaluate our representation and learning methods in a simulated environment, showing that our hierarchical representation is suitable for solving complex tasks, and that our state- and action-centric exploration biasing methods are both effective and complementary for efficiently learning AMDP transition functions. We show that the learned policy can be transferred to different tabletop organizational packing tasks, and validate that the policy can be realized on a physical system.**

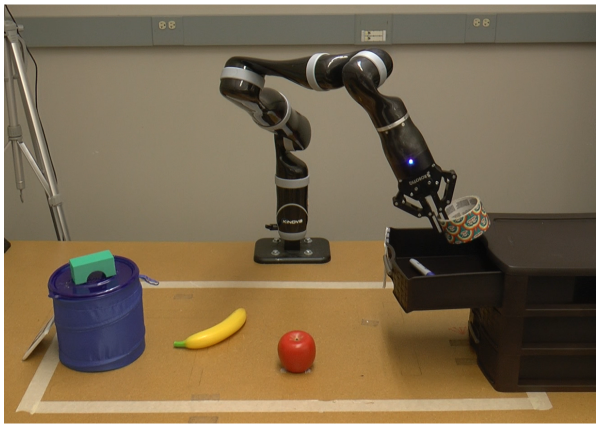
Fig. 1: A 4 item, 2 container (4I-2C) tabletop organization packing environment.

I. Introduction

Reinforcement learning (RL) is a promising paradigm for solving sequential decision problems, with the potential to allow robots to learn the effects of their actions directly through experience. However, traditional RL approaches face many scalability challenges in complex domains, particularly in the context of noisy or uncertain action effects. In this work, we show how the RL paradigm can be adapted in order to allow greater scalability, focusing specifically on domains in which an agent must modify the environment using highly non-deterministic actions.

Our approach builds on two existing Markov Decision Process (MDP) formulations that have been proposed to improve RL's applicability to robotics domains—Object-Oriented MDPs (OO-MDPs) [1] and Abstract MDPs (AMPDs) [2]. OO-MDPs reframe an MDP's components in terms of objects instantiated from a set of classes, representing object-object relations and learning transition models at the class level. The object-oriented approach improves transfer to environments with different numbers of objects, but it struggles with the large state spaces inherent to complex tasks. AMDPs extend OO-MDPs to a hierarchical representation that improves scalability by breaking problems into re-usable subgoals through state projection and abstract actions.

In this work, we introduce a tabletop organizational packing task, in which a robot must put sets of items away in organizational containers, using pick-and-place actions and performing more complex manipulation such as opening and closing containers. The combination of a) difficult-to-model stochastic action effects that occur from situating the problem on a robot physically interacting with its environment, and b) the combinatorial state-space explosion that occurs from adding new items to the environment, make this domain especially challenging and intractable to solve with traditional RL learning methods. We show performance of both OO-MDP and AMDP methods on this domain, and present techniques for leveraging human demonstration to facilitate the learning of complex policies. Specifically, the contributions of this work are: (1) formulation of a tractable hierarchical representation for organizational packing tasks, (2) development of a novel state- and action-centric demonstration-based method to bias exploration to learn transition functions, and (3) realization and evaluation of a full approach combining state- and action-centric demonstration representations to learn transition functions and execute AMDPs. We evaluate our approach in a simulated environment that represents the challenges inherent to learning these tasks on a robot, and then transfer the learned model to a physical robot. Our results show that our approach can effectively solve general tabletop organization and packing tasks, and that the learned model can transfer to task environments of varying complexity, as well as from simulation to a physical robot.

II. Related Work

A number of prior representations have been proposed that make RL more suitable to robotics domains. Object-Oriented MDPs [1] and relational MDPs [3] are two problem

*Georgia Institute of Technology, Atlanta, GA 30332, USA {dekent, siddhartha.banerjee, chernova}@gatech.edu

representations that formulate sequential decision problems in terms of objects instantiated from classes. Both methods use an object-oriented design to focus on relations between objects, which naturally capture important agent and environmental interactions in many problem domains, and allow for re-use of transition functions and rewards across domains with varying numbers of objects. We use OO-MDPs as a starting point for our approach, as they provide a simpler representation applicable to a single active agent, but OO-MDPs alone are not sufficient for our problem. While OO-MDPs are successfully used in simulated domains, they are less common in real-world systems (with a few exceptions such as [4], where the number of objects, actions, and relations are very low); despite the generalization provided by relations and classes, they become intractable in large state-action spaces.

Scalability of RL techniques has been improved through hierarchical approaches, which effectively reduce large state-action spaces by dividing complex tasks into independently solvable sub-problems. Abstract MDPs [2] extend OO-MDPs to a hierarchical representation. They create a hierarchy using abstract actions, each represented in turn by their own AMDPs, and use state projection functions to move through the hierarchy, controlling the size of the state space. We base our work on AMDPs, as they provide a hierarchical representation that includes the object-oriented environment modeling advantages of OO-MDPs. Other hierarchical task learning methods include hierarchical Q-learning [5], [6], [7], Semi-MDP planning with options [8], [9], and MAXQ [10]. Each have shown significant improvements in learning and planning time over flat representations, but present their own challenges for defining hierarchical structure and rewards. Hierarchical Task Networks present a non-RL hierarchical representation which can be learned from demonstration [11], [12], but are less suited to highly stochastic problems.

RL efficiency can also be improved by leveraging human input, such as by biasing exploration with policies derived from task demonstrations: a method often termed Learning from Demonstration (LfD) [13]. We first consider state-centric LfD approaches to sequential decision problems, which map states to actions. The Human-Agent Transfer (HAT) algorithm biases otherwise random exploration by extracting general rules from demonstrations using a depth-limited decision tree [14]. [15] provides further evidence that decision trees, even with shallow depth, can effectively learn policies for autonomous maze navigation from human demonstrations. Alternatively, LfD includes action-centric approaches which aim to reproduce common action sequences. Such approaches include plan networks for characterizing action sequences [16] and generating successor actions [17], Semi-MDP-dual graphs for representing ordering constraints over action sequences [18], and workflow-guided exploration [19] for creating state-agnostic action workflows to guide exploration for reinforcement learning over web forms. For non-sequential problems, [20] shows the advantages of guided self-exploration for affordance learning. We explore both state-centric exploration biasing inspired by HAT, and action-centric exploration biasing using a goal-centric implementation of plan networks in place of workflows, thereby adapting workflow-guided exploration to a robotics domain. Further, we integrate both methods in one algorithm, showing that state- and action-centric approaches provide complimentary exploration.

III. Problem Formulation

We begin by describing tabletop organization and packing as a high-level sequential decision problem, and discuss the problem's challenges. We then discuss our problem representation–an AMDP hierarchy built on a relation-based MDP formulation derived from OO-MDPs.

A. Task Description

We define a *tabletop organizational packing task* as an example of a high-level sequential decision task that can be completed by a robot. The goal of the task is to organize a set of pickable items by packing them in manipulable containers such as boxes and drawers. Different tabletop organizational packing tasks are represented by different environments, characterized by the number of items and the number of containers in the environment. An example environment with 4 items and 2 containers (4I-2C) is shown in Figure 1, where the goal is to sort fruits into the box and office supplies into the drawer. The robot can achieve the goal by executing a sequence of primitive actions, e.g. *pick*, *place*, and *move*, that are performed using grasp and place calculators, motion planning, and the robot's controllers.

Situating this task on a physical robot introduces challenges to what could otherwise be a traditional planning problem. First, the actions are stochastic. The robot's primitive actions do not always result in the desired behavior, where grasp, place, and motion planning failures can arise from object clutter, constrained spaces, perceptual noise, etc. Other actions involving pushing, pulling, or dropping objects involve difficult-to-model physical interactions between the robot's end effector and the objects, as well as between the objects themselves. Additionally, adding an object to the environment exponentially increases the size of the state space due to the combinatorial explosion of item and container positions. Further, when combining the size of the state space with the stochastic environment, the large state-action space introduces concerns of problem tractability. We address these concerns by using MDP representations based around object relations, which give some generalizability to the state, and hierarchical structures, which allow re-use of subgoals and transfer to other environments. We describe these representations below. We use the 4I-2C environment as a motivating task throughout this work, but our methods also transfer to other environments, as shown in Section V-D.

B. OO-MDP Formulation

In this section, we describe how tabletop organization can be formalized as an OO-MDP. OO-MDPs have the same components as traditional MDPs, i.e. $(S, A, P(s'|s, a), R(s))$, where S is the set of states of the environment, A is the set of actions the robot can take, P is a stochastic transition function used to generate new states given a state-action pair, and R is a reward calculated for a given state. OO-MDPs differ from traditional MDPs in that each of these components is reformulated based on a set of object classes C, each described by sets of attributes. The environment (including the agent) is fully described by a set of objects, $o \in O$, instantiated from C. Relationships between objects in the environment, and the agent itself, are defined as binary relations at the class level, where a relation $Rel(o_i, o_j)$ is a Boolean function of two objects' attributes. For more details, see [1].

For tabletop organizational packing, our classes include:

- **Item**: small graspable objects (to be put organized)
- **Box**: large, immovable storage containers
- **Lid**: lids to close boxes
- **Stack**: stacks of drawers
- **Drawer**: individual drawers in a stack
- **Gripper**: end-effector of the robot

All of our classes have (x, y, z) position attributes, Items have a label attribute, Boxes, Lids, Stacks, and Drawers have dimension attributes, and Grippers have an open/closed attribute. We ignore object orientation for Items, as our grasp action calculates grasps directly from object point clouds. As a simplification, we assume Boxes and Stacks have a fixed orientation, although our state could be extended by adding an orientation attribute to the Box and Stack class. Our set of object relations include:

- Relative positions: relations for *left_of*, *right_of*, *in_front_of*, *behind*, *above*, and *below*
- *touching*: a relation for object-object contact
- *closing*: a storage-specific relation representing if a Lid is closing a Box or if a Drawer is closing a Stack
- *holding*: a Gripper-specific relation indicating what object, if any, the gripper is currently grasping

We define a set of actions representing the robot's primitive actions. Each action is executable with a combination of grasp and place calculation, motion planning, and closed-loop control. The actions of our task include:

- *grasp(object)*: move to an object (with motion planning and collision detection) and close the gripper
- *place(object)*: move to a target object (with motion planning and collision detection) and release a held item
- *move(object **or** direction)*: straight-line movement in the xy plane that does not check for collisions, allowing the robot to push and pull objects
- *raise/lower*: move the gripper up or down
- *open/close*: open or close the gripper
- *reset*: move to a home position

As with a traditional MDP, we can solve for a policy with standard methods such as value iteration. The main advantage of OO-MDPs is that everything is defined at the class level, and thus once learned, the relations, transition models, and rewards can transfer to environments with different numbers of objects. Ideally, we can learn action effects for each class once, and transfer this knowledge to new environments, re-running value iteration to generate new policies. Unfortunately, in practice the OO-MDP formulation for the tabletop organizational packing task is intractable to solve even in the simplest case of storing 1 item in 1 container (1I-1C). This is due to the large proliferation of states and the combination of stochastic effects on all of the object attributes. We can, however, use the relations of the OO-MDP to increase the problem's tractability, as described in the next section.

Environment	**State-space size**		
	OO-MDP	Relation-based MDP	AMDP
1I-1C	1.13×10^{12}	9.45×10^{5}	9.47×10^{5}
2I-1C	4.54×10^{15}	1.98×10^{11}	9.47×10^{5}
2I-2C	5.10×10^{22}	4.22×10^{18}	1.89×10^{6}
3I-2C	1.84×10^{26}	2.32×10^{27}	1.89×10^{6}
4I-2C	6.43×10^{29}	6.72×10^{37}	1.89×10^{6}

TABLE I: State space growth for increasingly complex tabletop organizational packing environments containing M items to store in N containers (MI-NC). Unique state counts for the OO-MDP formulation are calculated using a 40x15x5 discretization of the tabletop workspace. State space sizes for the AMDP formulation are calculated as the sum of unique states over all of the AMDPs in the task hierarchy.

C. Relation-based Formulation

Since we cannot solve the OO-MDP for the 1I-1C case, we cannot benefit from its advantages for multiple objects. We instead develop a relation-based MDP formulation to preserve the advantages of object-oriented relations. By redefining the state as a vector of *only* relations, we significantly reduce the size of the state space while maintaining the relational information required to model the changes the robot can effect on the environment. This also generalizes across similar states, and discretizes the state space.

OO-MDPs represent transition functions as effects on object attributes. Since our new state does not directly contain the object attributes, we instead redefine the transition function to take the form $P([Rel_0(o'_0, o'_1), \ldots, Rel_l(o'_m, o'_n)] \mid [Rel_0(o_0, o_1), \ldots, Rel_l(o_m, o_n)], a)$, i.e. a direct mapping from relations to relations.

Even when using binary relations, the problem is still susceptible to combinatorial explosion from adding more objects. Table I shows the number of states for tabletop organizational packing environments under different representations with increasing numbers of objects. We note that all of these states are not necessarily reachable, and not all states may come up during execution, but the table still shows the trend of increasing state-space size. We find that our flat relation-based MDP formulation becomes intractable around the 2I-2C case. In the following section, we show how AMDPs can be used to improve scalability while maintaining the benefits of the relation-based representation.

D. AMDP Formulation

We extend our relation-based MDP formulation to AMDPs [2], creating a hierarchical representation of the

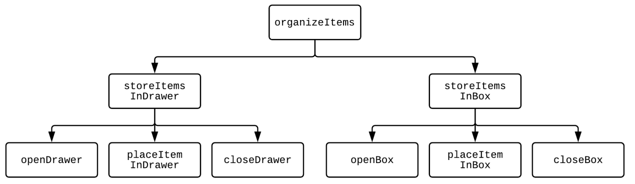

Fig. 2: 4I-2C AMDP hierarchy used for all experiments. All AMDPs at the bottom levels of each hierarchy include only primitive actions. Each AMDP takes an Item or a list of Items as a parameter to ground which Items are used in the state projection function $\tilde{F}(s)$.

tabletop organizational packing task. AMDPs have the same components as traditional MDPs, with two additions, resulting in the tuple[1] $(\tilde{S}, \tilde{A}, \tilde{P}(\tilde{s}'|\tilde{s}, \tilde{a}), \tilde{R}(\tilde{s}), \tilde{F}(s))$. First, the action set $\tilde{A}$ is extended to include abstract actions. Each abstract action represents a subgoal, which corresponds to another AMDP, thus creating a hierarchy. As an example, we define an AMDP for storing an item in a drawer with an abstract action set $\tilde{A} = \{\texttt{openDrawer}, \texttt{closeDrawer}, \texttt{placeItemInDrawer}\}$. We define the abstract actions each as AMDPs with primitive action sets containing all of the actions listed in III-B.

Second, AMDPs include a state projection function $F(s) \to \tilde{s}$, which projects the full state of the environment (in our case, the set of relations listed in Section III-B) to a set of elements relevant to the subgoal represented by the AMDP. For our example of storing an item in a drawer, the state projection functions for the `openDrawer` and `closeDrawer` AMDPs remove any relations not involving the Gripper and the Drawer objects, the projection function for the `placeItemInDrawer` AMDP removes any relations not involving the Gripper, the Item to be placed, and the Drawer objects, and the projection function for the `storeItemInDrawer` AMDP at the top of the hierarchy contains only two relations: a projection of the Item-Drawer spatial relations to a new relation *item_in_drawer*, and the *drawer_closing_stack* relation.

This formulation allows us to reduce any tabletop organizational packing task to the 1I-1C case. Better yet, we can reduce the problem to subgoals of the 1I-1C case, as with the store item in drawer example given above. Currently, the hierarchies are specified by hand. The full hierarchy for the 4I-2C case used in our experiments is shown in Figure 2. To solve the AMDP hierarchy for a policy, each AMDP can be solved independently using value iteration.

The transition functions for AMDPs at the lowest level of the hierarchy are difficult to specify, as they involve execution of the many primitive actions. We propose a method for learning these transition functions through exploration guided by demonstrations. AMDPs at the higher levels of our hierarchy (`organizeItems`, `storeItemsInDrawer`, and `storeItemsInBox`) contain only abstract actions and small states. As such, defining their transition functions by hand is trivial. Our approach learns the low-level transition functions, which combined with the handcrafted high-level transition functions and the AMDP hierarchy allow us to effectively solve each AMDP and generate task policies.

IV. Leveraging Demonstrations

We first describe the demonstrations required for learning the low-level AMDP transition functions, followed by a set of methods to learn said transition functions, and conclude with a complete method for generating policies from the AMDPs and demonstration data.

We perform transition function learning in a low-fidelity simulator that represents the characteristics of the physical environment[2]. We use a simulator to facilitate large-scale extensive evaluation. We perform final validation by transferring the learned models to the physical robot, with some refinement to account for differences not fully modeled in the simulator (see Section V-C for details).

A. Demonstrations

One of the core benefits of our approach is that, due to our hierarchical task decomposition, we do not need to collect demonstrations for the full 4I-2C task. Instead, we can simply collect demonstrations in 1I-1C environments and generalize those demonstrations to more complex tasks. For each demonstration, we log a list of projected state-action pairs $(\tilde{F}(s), a)$ for each AMDP. For our experiments, we collect full 1I-1C task demonstrations in one environment with a Drawer (training data for `openDrawer`, `placeItemInDrawer`, and `closeDrawer`) and one environment with a Box (training data for `openBox`, `placeItemInBox`, and `closeBox`).

B. Learning Transition Functions

Ideally, the robot could learn transition functions through exhaustive exploration in the simulator. In practice, the state space is too large to effectively learn the full task through random exploration alone. As an alternative, we leverage task demonstrations to learn transition functions through guided exploration. We learn general rules from the demonstrations to bias action selection, using either state-centric (similar to the HAT algorithm [14]) or action-centric (similar to workflow-guided exploration [19]) methods to explore the relevant state-action space. We describe the state- and action-centric methods below, followed by a description of the full exploration algorithm given in Algorithm 1.

For state-centric action selection biasing, we train a classifier on state-action pairs $(\tilde{F}(s), a)$ from the demonstration data for each AMDP, learning a mapping from states to actions. We selected a decision tree as the classifier because it has good recall for all of our actions, resulting in better coverage of the state-action space. We limit the depth of the decision tree to prevent overfitting. We also found that logistic regression and linear support vector machines (SVMs)

[1]Variables with tildes indicate components of an AMDP, to disambiguate them from components of a flat MDP.

[2]We provide the simulator, as well as our problem representations, methods, and experiments open source at `https://github.com/GT-RAIL/task_sim`

Algorithm 1 Transition Function Learning from Demonstration-Guided Exploration

```
Require: $demos$, $sim$, $train\_seeds$, $episodes$, $mode$
 1: $c \leftarrow$ classifier($demos.states$, $demos.actions$)
 2: $pn \leftarrow$ planNetwork($demos$)
 3: $T \leftarrow \{\}$
 4: $steps \leftarrow 0$
 5: $episode \leftarrow 0$
 6: $s \leftarrow sim$.reset($train\_seeds[episode]$)
 7: while $episode < episodes$ do
 8:     if rand() $< \epsilon$ then
 9:         if $mode$ is state-centric then
10:             $a \sim c$.predict($s$)
11:         else if $mode$ is action-centric then
12:             $a \sim pn$.successors($s$)
13:             if $a$ is $None$ then
14:                 $a \leftarrow$ randomAction()
15:     else
16:         $a \leftarrow$ randomAction()
17:     $s' \leftarrow sim$.execute($a$)
18:     $T$.update($s, a, s'$)
19:     $s \leftarrow s'$
20:     $steps \leftarrow steps + 1$
21:     if goalTest($s'$) or $steps \geq timeout$ then
22:         $episode \leftarrow episode + 1$
23:         $s \leftarrow sim$.reset($train\_seeds[episode]$)
24:         $steps \leftarrow 0$
25: return $T, c, pn$
```

performed well, and as such we include the three classifier variants in the evaluation presented in Section V-A. During exploration, we stochastically select actions proportional to the classifier's predicted probabilities over the full action set (Algorithm 1 line 11). Thus, the algorithm selects actions consistent with behavior observed for the current state.

For action-centric action selection biasing, we construct plan networks [16] for each AMDP, serving as workflows. The nodes of a plan network consist of tuples of $(preconditions, action, effects)$, which are determined from the demonstration action sequences. Nodes are connected by directed edges, representing tuples encountered in sequence in the demonstration data, with edge weights corresponding to the frequency that these sequences are encountered. While workflows are fully state-agnostic, plan networks incorporate state information as preconditions and effects. To generate more state-agnostic plan networks without fully eliminating key relation information, we use only the relations required for specifying the subtask goals as action preconditions and effects. To select an action (Algorithm 1 lines 13-15), we use the previous state, previous action, and current state to generate a node, localize that node within the plan network, and select an action stochastically (proportional to the edge weights) from the node's children whose preconditions are satisfied by the current state. If the current state of execution cannot be localized in the network,

Algorithm 2 AMDP Training and Task Execution

```
Require: $amdps, sim, demos, mode$
 1: for $amdp$ in $amdps$ do
 2:     $amdp.T, amdp.c, amdp.pn \leftarrow$ learnT($demos$)
 3:     $amdp.\pi \leftarrow$ valueIteration($amdp$)
 4: $s \leftarrow sim$.getState()
 5: while not goalTest(s) do
 6:     $amdp \leftarrow amdps$.root
 7:     $a \leftarrow amdp.\pi(amdp$.F$(s))$
 8:     while not isPrimitive($a$) do
 9:         $amdp \leftarrow amdps[a]$
10:         if $s$ in $amdp.\pi$ then
11:             $a \leftarrow amdp.\pi(amdp$.F$(s))$
12:         else
13:             if $mode$ is state-centric then
14:                 $a \sim c$.predict($s$)
15:             else if $mode$ is action-centric then
16:                 $a \sim pn$.successors($s$)
17:                 if $a$ is $None$ then
18:                     $a \leftarrow$ randomAction()
19:     $s \leftarrow sim$.execute($a$)
```

or if no children's preconditions are satisfied, we select a random action. Thus, the algorithm selects actions consistent with commonly observed action sequences.

The full exploration algorithm, presented in Algorithm 1, works as follows. For a given set of training environments parameterized by random seeds $train\_seeds$ and a given number of training episodes, the algorithm repeatedly selects and executes actions in our simulator (lines 8-26). As with collecting demonstrations, exploration need only occur in 1I-1C environments. Action selection is performed as a tradeoff between demonstration-guided exploration (lines 10-15) and random exploration (line 17), parameterized by probability ϵ. After executing the selected action and observing the results (line 18), the algorithm updates the transition function T, represented as a table indexed by state action pairs (s, a) which store a frequency table of resulting states s'. A training episode ends either when the goal is reached or an execution timeout is exceeded, the simulator is reset to the next training environment, and another episode begins (lines 22-26).

C. Executing AMDP Policies

We present the full algorithm for executing hierarchical AMDP policies in Algorithm 2. The algorithm begins with a training period for each AMDP (lines 1-3). Transition functions are learned using the method described in Section IV-B. We also store the classifier and plan network for each AMPD for later use. The algorithm then solves for the optimal (with respect to the learned transition functions) policy for each AMDP using value iteration.

With the individual policies learned, the AMDP hierarchy can generate primitive actions. This is accomplished by starting at the root of the hierarchy, the `organizeItems` AMDP, selecting the optimal action, and iteratively moving down the AMDP hierarchy selecting optimal actions until a primitive action is returned (lines 6-11). One downside to learning transition functions through exploration is that the algorithm may not learn a policy for every state.[3] In the case of an unseen state, we instead re-use the state- or action-centric rules to generate an action (lines 12-18), with the hope that it will either complete the task or guide execution back to a region of the state space known to the policy. Once a primitive action is selected, the algorithm executes the action and repeats this process, restarting at the top of the AMDP hierarchy, until the task is complete.

V. Experiments

For consistent evaluation of transition function learning methods and the performance of the full AMDP hierarchy, we establish a set of demonstrations, training environments, test environments, and an evaluation procedure. We first create a consistent set of 20 4I-2C training environments ($train\_seeds \in [0, 19]$). We next collect 20 demonstrations performed on 1I-1C tasks within the training environments, where the 1I-1C tasks are formed by reducing the training environments to a single Item and a single Container (with variants for Drawers and Boxes). The resulting demonstration data is then used, along with the training environments, to perform transition function learning using Algorithm 1 for each of the primitive-action AMDPs.

We perform evaluation on the training set by executing the full AMDP hierarchy for 5 runs in each training environment using Algorithm 2 to solve the full 4I-2C task, using the results to calculate a training environment success rate. We also create a consistent set of 100 heldout 4I-2C test environments ($train\_seeds \in [20, 119]$), on which we execute 1 run each with the full AMDP hierarchy to calculate a test environment success rate. For all training and testing evaluation runs, we enforce a 100 action limit, beyond which a run is considered a failure (an optimal solution takes approximately 20 actions). Evaluations were performed every 10 training episodes. In the following sections, we compare a set of exploration methods (Section V-A) and a set of reinforcement learning methods (Section V-B).

A. Exploration Method Evaluation

We evaluate three demonstration-guided exploration methods to compare the effects of state- and action-centric exploration biasing. The methods are as follows:

- State-Centric + Random (**SC**): Trading off using the state-action classifier and random action selection, this method represents purely state-centric exploration biasing. We perform evaluation using a decision tree, logistic regression, and an SVM as classifiers.
- Action-Centric + Random (**AC**): Trading off using the plan network and random action selection, this method represents purely action-centric exploration biasing.

[3]This can be mitigated with function approximation, but to date we have not had success, due to difficulties in abstracting the latent factors that govern transition dynamics (motion planning failures, etc.) from observation in our relation-based domain.

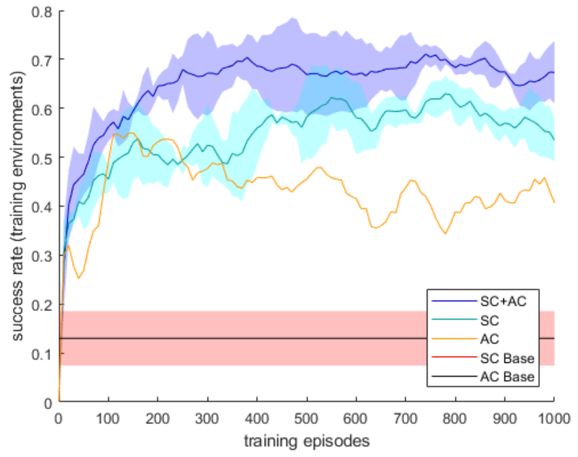

Fig. 3: Evaluation of exploration approaches for learning the 4I-2C task over the training environments. SC results are reported as the mean success rate achieved using the decision tree, logistic regression, and SVM state-action models, with shaded regions showing ±1 standard deviation. SC Base and AC Base performed equivalently with a 13% success rate; Rand omitted as it always produced a 0% success rate.

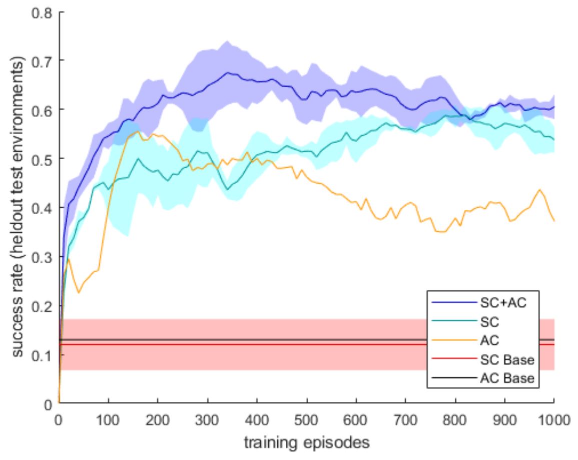

Fig. 4: Evaluation of exploration approaches for learning the 4I-2C task over 100 heldout test environments. Rand omitted as it always produced a 0% success rate.

- State-Centric + Action-Centric + Random (**SC+AC**): We also combine state- and action-centric exploration biasing to determine whether they are complimentary. To perform this method, instead of having Algorithms 1 and 2 select actions from either only the state-action mapping or only the plan network, the algorithms flip a coin at each loop iteration and use one approach or the other.

Additionally, we include a random exploration (**Rand**) baseline, to motivate the need for exploration-guided demonstration. Rand learns transition functions by always selecting a random action. Evaluation using the full model learned over 1000 training episodes resulted in a 0% success rate for both the training and testing environments. Random exploration has only a 3.9% success rate at solving the 1I-1C training environments during the exploration phase, and the policy learned from value iteration does not sufficiently cover the state-action space to solve the 4I-2C problem.

We also evaluate two non-exploration baselines, by using either a state-action classifier (**SC Base**) or a plan network (**AC Base**) directly for evaluation, without performing policy learning through transition function exploration. At evaluation time, the AMDPs directly use either the state-action classifier or the plan network, respectively, for action selection. As there is no exploration phase the methods' performances do not change over the training episodes.

The training environment success rates of all methods are shown in Figure 3, and the heldout test environment success rates are shown in Figure 4. Our results show that learning transition functions with demonstration-guided exploration and using them to solve for AMDP policies greatly improves the baseline results for all methods, in both the training environments and when generalizing to unseen environments.

Alone, AC reaches peak performance at 66% and 62% success in the training and test environments, respectively, and SC reaches peak performance at 75% and 68% success. When combined we see evidence that the state- and action-centric methods are complimentary. SC+AC outperforms all other exploration methods, reaching peak success rates of 88% in the training environments and 78% in the test environments. Of the state-centric classifiers, the decision tree was the most successful at generalizing to the test environments, with a 78% success rate vs. logistic regression's and SVM's 73% success rates for the SC+AC method.

B. Reinforcement Learning Comparison

With SC+AC established as our best exploration method, we evaluate its performance within various reinforcement learning approaches, shown in Figures 5 and 6. The approaches are as follows:

- Model-Based Demonstration-Guided Exploration (**SC+AC**): This approach directly uses Algorithm 1, as in the previous section, to learn the AMDP transition functions. This approach does not exploit the learned policy during transition function learning.
- Model-Based Demonstration-Guided Exploration with Exploitation (**SC+AC Exploit**): SC+AC with the additional tradeoff of exploiting the learned policy instead of exploring, in proportion to a decaying ϵ.
- Model-Free Demonstration-Guided Exploration (**SC+AC Q-learning**): A model-free implementation of Algorithm 1 that learns a table of Q-values rather than a transition function for each AMDP, trading off between exploration and exploitation of the learned policy in proportion to a decaying ϵ, where exploration selects either a random action or an action from either the state- or action-centric models.
- Model-Free Baseline (**Q-learning**): A standard implementation of Q-learning, where a table of Q-values is learned for each AMDP. As a baseline, exploration consists only of randomly selected actions.

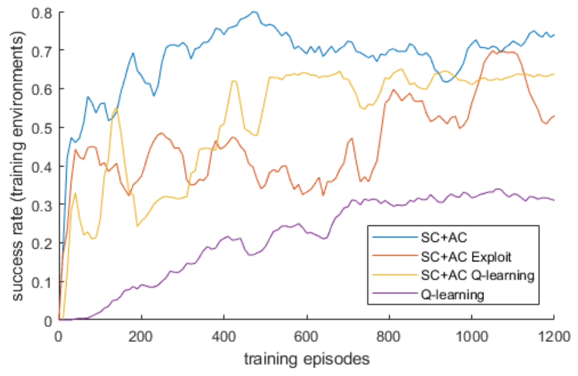

Fig. 5: Comparison of reinforcement learning approaches for the 4I-2C task on the training environments.

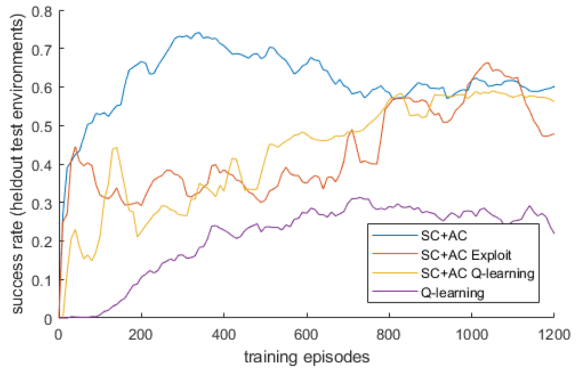

Fig. 6: Comparison of reinforcement learning approaches for the 4I-2C task on 100 heldout test environments.

Q-learning eventually converges to approximately a 50% training environment success rate and 40% test environment success rate after 5000 training episodes. Incorporating demonstration-guided exploration with Q-learning decreases convergence time by a factor of 5, with performance also increasing to 65% and 60% in the training and test environments, respectively. One of the fundamental downsides to using a model-free approach with an AMDP hierarchy, though, is that a separate Q-function must be learned for each AMDP. In contrast, learning transition functions allows for model re-use in AMDPs that use the same state projection function $\tilde{F}(s)$ and action set $\tilde{A}$ (e.g. for `openDrawer` and `closeDrawer`, or for `openBox` and `closeBox`). As such, the model-based approaches learn a successful policy more efficiently.

SC+AC without exploitation both converged fastest and reached the highest peak performance. This suggests that, for the 4I-2C environment, exploration around the policies learned from task demonstrations provides more useful information than exploiting the policy learned thus far. The result is consistent with the most common cause of task failure for all of our methods: when the robot reaches a state that it has not previously encountered, the state-action classifier or the

Method	Environment			
	2I-1C	3I-2C	4I-2C	5I-2C
SC+AC	0.781	0.557	0.638	0.474
SC	0.784	0.465	0.336	0.312
AC	0.814	0.563	0.482	0.458

TABLE II: Success rates from transferring the learned policy to task environments of varying difficulty. Success rates are calculated over 1000 instances of each environment type.

plan network i unable to guide the robot back to a known area of the state space, and the algorithm chooses actions blindly. To further improve performance, we require either additional exploration time, or function approximation that can generalize to unexplored states.

SC+AC converges after approximately 42000 total exploration action executions, which is relatively quick given the size of the state-action space. However, this is orders of magnitude too high to reasonably perform exploration using a physical robot. Therefore, we transfer our best model learned in simulation and refine it on the physical robot to perform our final validation.

C. Pysical Robot Validation

We transfer the full AMDP policy learned in simulation using the SC+AC method to the physical robot and environment shown in Figure 1. The task involves putting office supplies into a drawer and fruit into a box. The robot consists of a 7-DOF Kinova JACO, with a 2-finger Robotiq 85 gripper, and 3D perception from an Asus Xtion Pro camera. We assume that the environment is fully observable, and present the physical robot results as a proof of concept for transfer from simulation to a real environment. We leave the case of handling partially observable state to future work. We first refine the transition functions by executing the policy (with no exploration) for five iterations, updating the transition functions to correct for any major differences between our simulator and the real world. After re-running value iteration, the robot can execute the complete AMDP to solve a 4I-2C task. We show the task in its entirety in the included supplemental video[4].

D. Task Transfer

One of the main advantages of the AMDP representation is that the hierarchical structure allows for re-use of AMDPs at lower levels of the hierarchy to execute different high level tasks. We evaluate this by taking the same models trained and tested in Section V-A and executing the learned policies to complete tasks with varying numbers of items and containers. To further test generalization, we perform this evaluation over 1000 unseen testing environment seeds. The same policies can be used to complete both simpler and more complex tasks, as shown in Table II.

[4]Available in high quality at `https://youtu.be/11LB_wc5CGc`

VI. Conclusion

We have successfully applied a reinforcement learning approach to solving complex robot manipulation tasks with stochastic effects, by controlling the size of the state-action space with an AMDP task hierarchy built over an object-oriented relation-based state. The approach efficiently learns transition functions, using a novel demonstration-guided exploration algorithm that makes use of state- and action-centric demonstration representations, which we show to be complimentary. After learning a model once, our approach transfers to environments of varying complexity, and is fully realizable on physical systems.

Acknowledgement

This work is supported in part by NSF IIS 135580 and an Early Career Faculty grant from NASAs Space Technology Research Grants Program.

References

[1] C. Diuk, A. Cohen, and M. L. Littman, "An object-oriented representation for efficient reinforcement learning," in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 240–247.

[2] N. Gopalan, M. desJardins, M. L. Littman, J. MacGlashan, S. Squire, S. Tellex, J. Winder, and L. L. Wong, "Planning with abstract Markov decision processes," in *ICAPS*, 2017.

[3] C. Guestrin, D. Koller, C. Gearhart, and N. Kanodia, "Generalizing plans to new environments in relational MDPs," in *Proceedings of the 18th international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc., 2003, pp. 1003–1010.

[4] E. F. Morales and C. Sammut, "Learning to fly by combining reinforcement learning with behavioural cloning," in *Proceedings of the twenty-first international conference on Machine learning*. ACM, 2004, p. 76.

[5] P. Dayan and G. E. Hinton, "Feudal reinforcement learning," in *Advances in neural information processing systems*, 1993, pp. 271–278.

[6] T. D. Kulkarni, K. Narasimhan, A. Saeedi, and J. Tenenbaum, "Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation," in *Advances in neural information processing systems*, 2016, pp. 3675–3683.

[7] A. S. Vezhnevets, S. Osindero, T. Schaul, N. Heess, M. Jaderberg, D. Silver, and K. Kavukcuoglu, "Feudal networks for hierarchical reinforcement learning," *arXiv preprint arXiv:1703.01161*, 2017.

[8] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artificial intelligence*, vol. 112, no. 1-2, pp. 181–211, 1999.

[9] P.-L. Bacon, J. Harb, and D. Precup, "The Option-Critic Architecture," in *AAAI*, 2017, pp. 1726–1734.

[10] T. G. Dietterich, "Hierarchical reinforcement learning with the MAXQ value function decomposition," *Journal of Artificial Intelligence Research*, vol. 13, pp. 227–303, 2000.

[11] N. Nejati, P. Langley, and T. Konik, "Learning hierarchical task networks by observation," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 665–672.

[12] A. Mohseni-Kabir, C. Rich, S. Chernova, C. L. Sidner, and D. Miller, "Interactive hierarchical task learning from a single demonstration," in *Proceedings of the Tenth Annual ACM/IEEE International Conference on Human-Robot Interaction*. ACM, 2015, pp. 205–212.

[13] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robotics and Autonomous Systems*, vol. 57, no. 5, pp. 469–483, may 2009.

[14] M. E. Taylor, H. B. Suay, and S. Chernova, "Integrating reinforcement learning with human demonstrations of varying ability," in *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*. International Foundation for Autonomous Agents and Multiagent Systems, 2011, pp. 617–624.

[15] C. Crick, S. Osentoski, G. Jay, and O. C. Jenkins, “Human and robot perception in large-scale learning from demonstration,” in *Proceedings of the 6th international conference on Human-robot interaction*. ACM, 2011, pp. 339–346.

[16] J. Orkin and D. Roy, “The restaurant game: Learning social behavior and language from thousands of players online,” *Journal of Game Development*, vol. 3, no. 1, pp. 39–60, 2007.

[17] ——, “Automatic learning and generation of social behavior from collective human gameplay,” in *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*. International Foundation for Autonomous Agents and Multiagent Systems, 2009, pp. 385–392.

[18] B. Hayes and B. Scassellati, “Online development of assistive robot behaviors for collaborative manipulation and human-robot teamwork,” in *Proceedings of the Machine Learning for Interactive Systems(MLIS) Workshop at AAAI*, 2014.

[19] E. Z. Liu, K. Guu, P. Pasupat, T. Shi, and P. Liang, “Reinforcement Learning on Web Interfaces Using Workflow-Guided Exploration,” *arXiv preprint arXiv:1802.08802*, 2018.

[20] V. Chu, T. Fitzgerald, and A. L. Thomaz, “Learning object affordances by leveraging the combination of human-guidance and self-exploration,” in *Human-Robot Interaction (HRI), 2016 11th ACM/IEEE International Conference on*. IEEE, 2016, pp. 221–228.