Learning value functions with relational state representations for guiding task-and-motion planning

Beomjoon Kim **Luke Shimanuki**
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
{beomjoon,lukeshim}@mit.edu

Abstract: We propose a novel relational state representation and an action-value function learning algorithm that learns from planning experience for *geometric task-and-motion planning* (GTAMP) problems, in which the goal is to move several objects to regions in the presence of movable obstacles. The representation encodes information about which objects occlude the manipulation of other objects and is encoded using a small set of predicates. It supports efficient learning, using graph neural networks, of an action-value function that can be used to guide a GTAMP solver. Importantly, it enables learning from planning experience on simple problems and generalizing to more complex problems and even across substantially different geometric environments. We demonstrate the method in two challenging GTAMP domains.

Keywords: Task and motion planning, value-function learning

1 Introduction

Robots that can manipulate objects in large, complex arrangements must solve difficult *task and motion planning* (TAMP) problems that integrate concerns ranging from high-level task achievement (managing a household or searching through rubble for disaster victims) to low-level joint-motion trajectories. TAMP problems are difficult to solve because they are in a hybrid discrete-continuous space, so neither discrete-space planning methods from the AI literature nor continuous-space optimization or sampling-based motion-planning methods are individually directly applicable. Although TAMP problems may include a variety of domain-specific aspects, such as cooking or manufacturing, they all by definition contain geometric sub-problems, which require the manipulation of objects and motion of the robot to achieve goals defined in terms of a sequence of relative or absolute positions of objects and the robot. We will call problems involving purely geometric object-manipulation goals *geometric TAMP* (GTAMP) problems (they include navigation among movable obstacles (NAMO) [1] and rearrangement [2] problems, among others), and focus on them for concreteness in this paper, though the techniques developed here have a direct extension to more general TAMP problems. Figure 1 illustrates two GTAMP example problems.

One promising approach to GTAMP is to perform a forward search from the current state, first branching on the choice of an *abstract action*, which specifies the action type and discrete parameters (such as *pick object a*), but does not yet specify continuous parameters (such as the grasp to be used and the object pose). Then, given the choice of an abstract action, the problem of selecting the continuous parameters can be handled using random sampling. A critical difficulty with this approach is search guidance in the discrete space: in domains of interest, with large numbers of objects and action types, the branching factor of the search is very large, making the search intractable without guidance.

It has been demonstrated in many domains, most vividly by AlphaZero [3], that methods inspired by reinforcement learning can be used to learn search-control information. In this paper, we explore the use of learned action-value functions for speeding up the discrete part of the search in GTAMP. Although learning approaches can also be used to speed up continuous action choices [4, 5, 6], we expect that there is a much greater opportunity for generalization across substantially different problem instances for predicting abstract-action values.

3rd Conference on Robot Learning (CoRL 2019), Osaka, Japan.

The fundamental difficulty of learning an abstract-action value function for GTAMP is representational: a GTAMP planner typically applies to a wide range of environments involving different three-dimensional arrangements of a robot, fixed obstacles (walls, tables) and movable objects (some of which are intended to be moved, others of which may be moved if necessary), potentially over a large and variably sized spatial extent. We must design a representation for learning that can generalize over this very wide range of problem-instance variability. In this work, we assume that the world state (shapes and poses of objects and robot configurations) is known in advance, or has been estimated via a prior process. Representations traditionally used in machine learning cannot easily capture all the information in a GTAMP problem state: fixed-length vectors do not effectively represent domains with potentially widely varying numbers of objects.

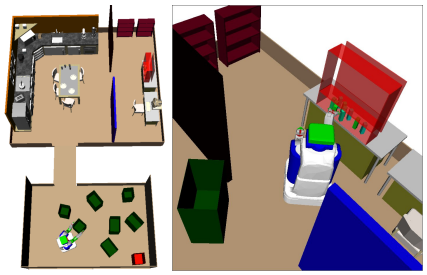


Figure 1: Left: moving the red box into the kitchen, while keeping the other boxes in the current room. Right: packing small objects from the red cupboard into the green box. Movable objects are randomly placed to construct problem instances.

We observe that, for the purposes of guiding the search for abstract actions, and especially when our objective is to generalize aggressively over environments, it is not necessary to represent the state in complete geometric detail. Instead, we wish to capture the essential geometric aspects of a scenario, but encode them abstractly in a way that affords generalization. For these reasons, we develop an abstract representation of states and goals, which consists of a set of relational terms representing a property of an object or a relation among a set of of objects. A value function can then be encoded using a graph neural network (GNN), which is parameterized by a fixed set of weights, but can be applied to state representations with any number of objects. Goals for GTAMP problems can also be specified relationally, and can be similarly encoded in the GNN.

We present a novel set of geometric predicates, which are used to construct the abstract state representation. They are designed to capture a fundamental difficulty of GTAMP problems, which is the need to move some objects in order to enable the movement of others. Thus, for example, we define a predicate $\textsc{OccludesManip}(o_1, o_2, r)$, which means that object o_1 is in the swept volume needed by the robot to manipulate object o_2 into region r. These predicates are, emphatically, not sufficient for making a complete, detailed plan, but they are sufficient for encoding effective search control information at the abstract level. They take advantage of the substantial generality and robustness of robot motion-planning algorithms, which are used to compute their values in a state.

We train a value function based on planning experience of solving small problem instances, using a combination of mean-squared-error and ranking loss. Because our abstract representation is general and captures the essential geometric features of each problem instance, and because the GNN structure abstracts over the combinatoric (objects, regions, and action types) aspects of the problem, learning a value function using this representation is very sample efficient, and it allows us to train the value function on experience from easily solvable problem instances, but apply it to much larger and more difficult instances. We demonstrate that the method significantly speeds up the search for solutions to complex problems, outperforming several existing approaches to GTAMP problems.

2 Related work

Our proposed method can be seen as doing a type of imitation learning from an oracle in the form of an existing, more limited, planner. Our key contribution is the approach for representing and learning a value function using a message-passing graph neural network (GNN) defined over geometric predicates of the state that defines the preconditions for the abstract actions. Although there has been previous work on learning to act using relational representations, aimed at increased generalization across environments [7, 8], it has not addressed realistic motion planning constraints.

Learning to guide planning Our work is related to the substantial body of work on learning to guide discrete planning. The most closely related work learns a heuristic function on states, to be used in planners based on heuristic search. This is typically formulated as supervised learning from planning

experience on related problems. Learning the heuristic function directly has proven challenging. Perhaps the most successful of these methods [9] learns domain-dependent corrections to an existing domain-independent heuristic. Many approaches learn how to best combine a variety of domain independent heuristics [10, 11]. Other approaches [12] learn how to rank actions directly. One of the algorithms in [13], like ours, aims to learn a Q-value function for ranking actions; however, it requires training data that provides the Q-values of sub-optimal actions.

The closest work to ours [14] also addresses TAMP problems and learn Q-values for high-level planning using a similar margin-based loss. However, their search is over plan refinements, rather than abstract domain actions and they provide no guidance on the choice of state representation. Bhardwaj et al. [15] aim to learn a heuristic to guide greedy search in grid-based motion-planning problems. They also learn Q-values by imitating an oracle (a standard graph search) but they exploit the fact that they can easily get many training examples by solving all-sources shortest path problems. Also, a state of their search is specified by a set of user-defined features on the whole state of the search, including the search queues. Our work aims to provide an approach for defining features for the challenging geometric states that arise in TAMP problems. There is a body of related work on guiding the choice of continuous parameters given the abstract operator sequence [5, 6, 4] which can be readily combined with our work within a single system.

Graph Neural Networks Graph neural networks [16, 17, 18] (see surveys of Battaglia et al. [19], Zhou et al. [20], Wu et al. [21]) incorporate a *relational inductive bias*: a set of entities and relations between them. In particular, we build on the framework of *message-passing neural networks* (MPNNs) [22], similar to *graph convolutional networks* (GCNs) [23, 19]. A key advantage of GNNs is that they learn a fixed-size set of parameters from problem instances with different numbers of entities. After learning, the GNN can be applied to arbitrarily large sets of objects and relations. This is crucial for GTAMP problems where the number of objects varies widely.

3 Problem formulation

A *geometric TAMP* (GTAMP) problem is specified by: a set of fixed rigid objects $\mathbf{O}^{(F)} = \{o_i^{(F)}\}_{i=1}^{n_1}$, a set of movable rigid objects $\mathbf{O}^{(M)} = \{o_i^{(M)}\}_{i=1}^{n_2}$, and a set of workspace regions $\mathbf{R} = \{r_i\}_{i=1}^{n_3}$, whose poses are defined relative to a *parent* object, which can be a movable object such as a tray, or a fixed object, such as a table. A state of the system is determined by the poses $\mathrm{P}_{o_i^{(M)}}$ of the movable objects and the configuration $c \in \mathcal{C}$ of the robot, so $s = (\mathrm{P}_{o_1^{(M)}}, \cdots, \mathrm{P}_{o_{n_1}^{(M)}}, c)$. All objects and regions have fixed known shapes.

The action space consists of a set of n_o operations $\mathcal{O} = \{\mathfrak{o}_1, \cdots, \mathfrak{o}_{n_o}\}$ that can cause collision-free motion of a single object. The robot might have one or more such operations available, such picking-and-placing, pushing, or throwing. Each operation is specified with a fixed number of operation-specific continuous parameters $\kappa \in K_{\mathfrak{o}}$, such as a target pose or a grasp, and discrete parameters $\delta \in \mathbf{O}^{(M)} \times \mathbf{R}$, specifying which object to move and a region to move it into. Thus, an operator instance $\mathfrak{o}(\delta, \kappa)$ for $\mathfrak{o} \in \mathcal{O}$ is a concrete operation that can be executed by the robot. Associated with it is a low-level robot motion expressed as a sequence of configurations, which is obtained by calling a motion planner. We will focus on *abstract actions*, which have the form $\mathfrak{o}(\delta)$, and represent operations whose discrete parameters such as objects and regions are determined, but whose continuous parameters will be selected later. We write δ to refer to an element of $\mathbf{O}^{(M)} \times \mathbf{R}$, but sometimes use (o, r) for clarity.

Each operation $\mathfrak{o}(\delta, \kappa)$ induces a mapping $T(\cdot, \mathfrak{o}(\delta, \kappa))$ from a world state s, in which it is executed, to a resulting world state $s' \in \mathcal{S}$. If the operation cannot be legally executed in s, we can let $s' = s$ or an absorbing "failure" state. In any given GTAMP problem, the objective is to find a sequence of concrete actions for the robot that will cause a sequence of state changes in the environment so that it enters some state S_G in a set $\mathcal{G} \subset \mathcal{S}$ of *goal states*. The *value* of an action trajectory of length t that reaches a goal state is $-t$; the *value* of a trajectory that never reaches a goal state is $-\infty$. This value function corresponds to a deterministic MDP-like model, in which each action has a reward of -1, except that when a goal state is reached, the system transitions to an absorbing state with reward 0 for all actions. This formulation can be easily extended to more general reward functions where, e.g., operations do not all have the same cost.

A *GTAMP planning problem* is characterized by $(\mathbf{O}^{(M)}, \mathbf{O}^{(F)}, \mathbf{R}, s_0, \mathcal{O}, \mathcal{G}, T)$, where $(\mathbf{O}^{(M)}, \mathbf{O}^{(F)}, \mathbf{R})$ defines the *environment*, s_0 is the initial state, $\mathcal{O}$ specifies the set of operators, $\mathcal{G}$ specifies the goal set and T specifies a deterministic transition function. Given a planning problem, we define the *Q-value function*, $Q(s, \mathfrak{o}(\delta, \kappa))$ as

$$Q(s, \mathfrak{o}(\delta, \kappa)) = \begin{cases} 0 & \text{if } s \in \mathcal{G} \\ -1 + \max_{\mathfrak{o}', \delta', \kappa'} Q(s', \mathfrak{o}'(\delta', \kappa')) & \text{otherwise} \end{cases},$$

where $s' = T(s, \mathfrak{o}(\delta, \kappa))$. In general, the Q function for detailed operator specifications, including continuous parameters, is very difficult to represent and does not generalize well across environments. We will focus on the *abstract Q-value function*, denoted Q_a, which is defined as

$$Q_a(s, \mathfrak{o}(\delta)) = \begin{cases} 0 & \text{if } s \in \mathcal{G} \\ -1 + \sup_{\kappa} \max_{\mathfrak{o}', \delta'} Q_a(s', \mathfrak{o}'(\delta')) & \text{otherwise} \end{cases},$$

where, again, $s' = T(s, \mathfrak{o}(\delta, \kappa))$, which is the next state assuming the best choice of κ in this context. Our objective will be to learn Q_a from data, and to use it to guide the search for plans in novel environments. We now describe the representation and learning algorithm for abstract Q-functions.

4 Representing and learning an abstract Q function

Our objective is to learn an abstract Q-value function that generalizes effectively from small to large problem instances, and across environments, with little training data. To do so, we design an abstract representation of both states and goals, and use it to form a network that can be trained on a small amount of data. It is important to note that our representation needs to be sufficiently detailed to reliably select appropriate abstract actions, but not as detailed as would be necessary to make a detailed geometric plan. Given this representation, we construct a graph neural-network, which allows us to learn a finite parameterization of a model that applies to problems with varying horizons, by using planning experience.

Relational state and goal representation We specify a goal set $\mathcal{G}$ as a conjunction of statements of the form $\textsc{InRegion}(o, r)$, where $o \in \mathbf{O}^{(M)}$ and $r \in \mathbf{R}$, which are true if o is contained entirely in region r. It is also possible to specify the final robot configuration as part of the goal for NAMO problems, or final poses of objects for specifying rearrangement problems.

We assume that each operator manipulates a single object to move it into a region or to a different pose within its region, so the preconditions of executing $\mathfrak{o}(\delta, \kappa)$ can be characterized by two volumes of workspace, $V_{\text{pre}}(q, \mathfrak{o}(\delta, \kappa))$ and $V_{\text{manip}}(\mathfrak{o}(\delta, \kappa))$, which must be clear of obstacles before the operation can be executed. The volume $V_{\text{pre}}(q, \mathfrak{o}(\delta, \kappa))$, where $\delta = (o, r)$, is the swept volume that the robot must move through, from its current configuration q to a configuration q' in which o can be reached. The volume $V_{\text{manip}}(\mathfrak{o}(\delta, \kappa))$ is the swept volume that the robot and object must move through, from their configurations at the beginning of the operation (P_o, q'), to their configurations at the end of the operation, as determined by continuous parameters κ.

We construct a relational abstract representation of the state $s = (\text{P}_{o_1^{(M)}}, \ldots, \text{P}_{o_n^{(M)}}, q)$ and goal $\mathcal{G}$, denoted $\alpha(s, \mathcal{G})$, as a conjunction of all true instances of the following relations, applied to *entities* $e \in \mathbf{O}^{(M)} \cup \mathbf{R}$: $\textsc{IsRegion}(e)$, true if e is a region; $\textsc{IsObject}(e)$, true if e is an object; $\textsc{IsGoal}(e)$, true if e is mentioned in the goal specification $\mathcal{G}$; $\textsc{InRegion}(o, r)$, true if object o is currently in region r; $\textsc{PreFree}(o)$, true if $\exists \kappa$ such that $V_{\text{pre}}(q, \mathfrak{o}((o, r), \kappa))$ is collision-free; $\textsc{ManipFree}(o, r)$, true if $\exists \kappa$ such that $V_{\text{manip}}(\mathfrak{o}((o, r), \kappa))$ is collision free; $\textsc{OccludesPre}(o_1, o_2)$, true if o_1 is an object that overlaps the swept volume $V_{\text{pre}}(q, \mathfrak{o}((o_2, r), \kappa))$, where κ is chosen to avoid collisions if possible; and $\textsc{OccludesManip}(o_1, o_2, r)$, true if o_1 is an object that overlaps the swept volume $V_{\text{manip}}(\mathfrak{o}((o_2, r), \kappa))$, where κ is chosen to avoid collisions if possible. The detailed implementations for the last four relations are specific to an operator *class*, such as pick-and-place or pull. The value of any of these predicates, if applied to arguments that are clearly the wrong type, is false. A critical aspect of this representation is that it also encodes the goal, which allows us to learn a single Q value function which can apply to any state-goal pair.

Given a concrete state s and goal $\mathcal{G}$, we must compute values for all instances of these predicates in that domain. The last four require non-trivial computation, including finding feasible κ values and computing the motion plans to obtain the necessary swept volumes. Ideally, we would find κ and

associated trajectories so that V_{manip} and V_{pre} had a minimum number of collisions with obstacles in the world, which can be very costly in the general case[24]. We compute them only approximately, in two stages: first, we attempt to find a collision-free κ and trajectory; if that fails, then we simply find κ and trajectory that are collision-free with respect to the fixed obstacles, but may collide with movable obstacles. We extensively cache collision checks, continuous parameters, and paths to make the repeated computation of the predicates efficient (details in the supplementary material).

Abstract Q-value function representation For each operator $\mathfrak{o}$, we define a graph neural network (GNN) $\widehat{Q}_{\mathfrak{o}}(\alpha(s,\mathcal{G}),\delta)$ that approximates $Q_a(s,\mathcal{G},\mathfrak{o}(\delta))$ for all $\delta \in \mathbf{O}^{(M)} \times \mathbf{R}$, $s \in S$. We begin by describing the input to the network, which is an encoding of the relations in $\alpha(s,\mathcal{G})$: for all entities e_i, x_i is a vector of unary predicate values ($\textsc{IsObject}(e_i)$, $\textsc{IsRegion}(e_i)$, $\textsc{IsGoal}(e_i)$, $\textsc{PreFree}(e_i)$); for all ordered pairs of entities e_i, e_j, x_{ij} is a vector of binary predicate values ($\textsc{InRegion}(e_i,e_j)$, $\textsc{OccludesPre}(e_i,e_j)$, $\textsc{ManipFree}(e_i,e_j)$); for all entities e_i, e_j and regions r_k, x_{ijk} is a single ternary relation value ($\textsc{OccludesManip}(e_i,e_j,e_k)$). We begin by computing, for each (e_i,e_j,r_k) triple, an embedding of the input information relevant to that triple, using a feed-forward neural network with parameter set θ_1, denoted by function f:

$$c_{ijk} = f(x_i, x_j, x_k, x_{ij}, x_{ik}, x_{jk}, x_{jik}; \theta_1) \ .$$

We compute two hidden-state vectors of the GNN for each object i and region k pair. The hidden-state values are initialized as $u_{ik}^{(0)} = f(x_i,\theta_2)$ and $v_{ik}^{(0)} = f(x_i,\theta_3)$, using fully connected neural networks with parameters θ_2 and θ_3. On each iteration, we compute a message, m_{ijk}, for each (e_i,e_j,r_k) triple, which can be interpreted as a message from entity i to the *pair* of entity j and region k: $m_{ijk}^{(t+1)} = f(u_i^{(t)}, v_j^{(t)}, c_{ijk}; \theta_4)$. This message has the same dimension as x_i. We aggregate these messages using averaging, so $m_{jk}^{(t+1)} = \frac{1}{n}\sum_i m_{ijk}^{(t+1)}$. The new hidden-state vectors are computed as $u_{jk}^{(t+1)} = f(m_{jk}^{(t+1)};\theta_2)$ and $v_{jk}^{(t+1)} = f(m_{jk}^{(t+1)};\theta_3)$. After T rounds of message passing, we compute the value of object o and region r as $\widehat{Q}_{\mathfrak{o}}(\alpha(s,\mathcal{G}),(o,r);\theta) = f(m_{o,r}^{(T)};\theta_5)$. Ultimately, this network can be seen as computing a regression from the abstract relational state and goal description to the value of a proposed action $\mathfrak{o}(\delta)$, parameterized by $\theta := (\theta_1,\theta_2,\theta_3,\theta_4,\theta_5)$.

Learning from planning experience Finally, we describe the loss function and training strategy for determining parameters θ given data. Because we know the transition model for the domain, we can find feasible plans for small problem instances efficiently using existing planning algorithms.

From each such plan, which is a sequence of state-action pairs $[(s_0,\mathfrak{o}_0(\delta_0,\kappa_0)),\ldots,(s_{T-1},\mathfrak{o}_{T-1}(\delta_{T-1},\kappa_{T-1})),(s_T,None)]$ in which $s_T \in \mathcal{G}$, where $\mathcal{G}$ is the goal for which that plan was made, we can construct $T+1$ supervised training examples of the form $(s_t,\mathfrak{o}_t(\delta_t),q)$, where $q = -(T-t)$, which denotes the Q-value of using the abstract action in state s_t. Recall that the goal for which each plan was constructed is indicated in each s_t of the sequence. Aggregating this data from multiple start-goal pairs, and partitioning it according to the abstract action types $\mathfrak{o}$, we end up with a data set $\mathcal{D}_{\mathfrak{o}}$ for each $\mathfrak{o}$, with entries of the form $(s,\mathcal{G},\delta,q)$.

Given a parametric model in the form of a GNN and a dataset $\mathcal{D}_{\mathfrak{o}}$, we adjust parameters θ to minimize a loss function. The most straightforward approach would be to treat this directly as a regression problem, and penalize squared error on the predicted Q value, with a squared error loss. But, because our training data is gathered from successful planning problem instances, we have no examples of state-action pairs with very small or negative-infinite q-values (indicating a dead-end), so we are only covering part of the relevant input space. However, to do a good job of action selection, it is more critical that the Q function assign a higher value to the best action in a state than it does to the other actions [25]. For this reason, we augment the squared-error loss with a large-margin term that encourages the model to assign a higher value to the selected abstract action than the values of other actions in the given state:

$$\mathcal{L}_{LM}(\theta) = \sum_{(s,\mathcal{G}\delta,q)\in\mathcal{D}_{\mathfrak{o}}} (\widehat{Q}_{\mathfrak{o}}(\alpha(s,\mathcal{G}),\delta;\theta) - q)^2 + \lambda \max(0, 1 - M_Q(\alpha(s,\mathcal{G}),\delta;\theta)) \ \text{, where}$$

$$M_Q(\alpha(s,\mathcal{G}),\delta;\theta) = \widehat{Q}_{\mathfrak{o}}(\alpha(s,\mathcal{G}),\delta;\theta) - \max_{\delta'\in\Delta\setminus\{\delta\}} \widehat{Q}_{\mathfrak{o}}(\alpha(s,\mathcal{G}),\delta';\theta) \ .$$

The second term evaluates to zero if M_Q, the margin between the value of the action selected in the training solution and the best predicted value among the other actions, is greater than 1.

5 Sampling-based heuristic-search algorithm for GTAMP

The key distinction between GTAMP problems and most graph-search problems is that the feasibility of a transition is very expensive to evaluate: to check whether an operation is feasible, it is necessary to sample feasible values of its continuous parameters and call an inverse-kinematics solver and a motion-planner to ensure the existence of a robot configuration and a collision-free path for performing the operation. So, instead of the traditional state-based search, in which a state is expanded and its successors are added to the queue, we maintain a priority queue of *state-and-abstract-action pairs*, which we call *abstract edges*. We then combine the heuristic value of the state and the ranking of the abstract action from the Q-values to determine which abstract edge to explore first.

Algorithm 1 SAHS($s_0, \mathcal{G}, N_{\text{smpl}}, \{\widehat{Q}_{\mathfrak{o}}\}_{\mathfrak{o}\in\mathcal{O}}, H$)

```
queue = PriorityQueue()
for δ ∈ O^(M) × R, o ∈ O
    queue.add((s_0, o(δ)), p(α(s, G), o(δ)))
while not time_limit_reached
    s, o(δ) = queue.pop()
    κ = UniformSmplCont(s, o(δ), N_smpl)
    if κ is feasible
        s' = T(s, o(δ, κ))
        s'.path = s.path + o(δ, κ)
        if s' ∈ G
            return s'.path
        for δ' ∈ O^(M) × R, o ∈ O
            queue.add((s', o(δ')), p(α(s, G), o(δ)))
    if queue.empty
        for δ ∈ O^(M) × R, o ∈ O
            queue.add((s_0, o(δ)), p(α(s, G), o(δ)))
```

We note that, while our learning objective allows us to approximate the cost-to-go, this information is in general more accurately captured by the heuristic function, especially if our Q-function is trained on easier problems (ex. shorter-horizon problems) and applied to harder problems. What we hope to get from the Q-function is the ranking information among the abstract actions within a state. To benefit from both heuristic and ranking information, we define our *priority function* to be

$$\mathbf{p}(\alpha(s,\mathcal{G}),\mathfrak{o}(\delta)) = -H(\alpha(s,\mathcal{G})) + \lambda \cdot \frac{exp\left(\widehat{Q}_{\mathfrak{o}}(\alpha(s,\mathcal{G}),\delta;\theta)\right)}{exp\left(\sum_{\mathfrak{o}',\delta'} \widehat{Q}_{\mathfrak{o}'}(\alpha(s,\mathcal{G}),\delta;\theta')\right)}$$

where λ controls how much we weigh the Q-function. The priority of an abstract edge $(\alpha(s,\mathcal{G}),\mathfrak{o}(\delta))$ is higher if the heuristic value of the state is low and the Q-value of the abstract action is high. This form simplifies tuning of λ.

Algorithm 1 defines sampling-based abstract-edge heuristic search (SAHS), which is a greedy strategy with respect to the priority function. In addition to the components needed to specify a GTAMP problem, it also takes the number of attempts to sample a feasible set of continuous parameters, N_{smpl}, the set of learned Q-value functions for each operator, and a heuristic function H that can be applied to a state. At each iteration, the algorithm selects the abstract edge with the highest priority, and attempts to construct a successor state by sampling feasible continuous parameters for the abstract action in the associated state. If this is successful, then abstract edges based on the new state are added to the queue. If the next state is in the goal set, it returns the plan by retracing the path to the root. If it fails to sample feasible continuous parameters, it moves onto the next tuple on the queue. Unlike discrete graph search, our search space involves continuous values, so when the queue is empty we add the initial abstract edges back to the queue.

6 Experiments

We now evaluate our learning algorithm and representation by applying it to challenging GTAMP problems. Our experiments are intended to support two claims: (1) the Q-value function learned from problems in a single environment can generalize to different environments and to larger and more difficult problems without re-training, and (2) the Q-value function learned by our approach is more effective at guiding the search than several reasonable baselines.

To evaluate generalization, we generate planning experience in the box-moving domain shown in Figure 1 (left) for problem instances in which the goal is to move a single box, shown in red, to

the kitchen region at the top. Problem instances differ in the initial pose of the robot and those of the objects. To evaluate its generalization capability to larger problems, we test whether the learned Q-value function, without retraining, is effective in guiding the search for packing 4 boxes into the kitchen region. To evaluate its generalization capability across different environments, we test the Q-value function, again without retraining, in the cupboard environment shown in Figure 1 (right), in which the objective is to pack small objects from the red cupboard into the green box. We test baseline approaches using the same training data and test problems.

We collect training data by solving many instances of single-box moving problems in the box-moving environment. To solve these problems, we use the Resolve Spatial Constraint (RSC) method proposed in [1]. This algorithm aims to move a single object into a desired region while moving any obstacles out of the way. The algorithm limits itself to touching each object at most once. We take the plan given by RSC, each step of which indicates an object to be moved and a region to which the object should be moved. For each state, we compute our predicates and pair them with the action taken in that state and the estimated Q-value of taking that action in that state, computed as the negative of the number of remaining steps in the plan. This state, action, and the negative remaining steps constitutes our training data.

We have the following learning and non-learning benchmarks:

1. I-RSC: Iterative RSC. In I-RSC, we extend RSC to the multi-object case by first planning without finding full motion plans, but still checking for collision-free object placements and initial and final robot configurations. From this, we get an order to pack objects into the goal region. Each single-object packing sub-problem is solved by an application of RSC. If, after some number of iterations, RSC does not find a solution, we modify the object ordering and try randomly permuting the unplaced boxes. The algorithm will eventually try all orderings if given enough time.

2. SAHS-HCOUNT: SAHS with a hand-designed state-based-heuristic function that uses the geometric predicate state representation, and without a learned Q-function (so the priority function consists only of the heuristic function). It recursively counts the number of objects that need to be cleared from the pick-and-place motions for goal objects. More formally, $H_{count}(s, \mathcal{G}, o, r) = |M|$, where $M = \{o' \mid \exists r' \text{ s.t. } (o', r') \in \mathcal{G} \text{ or } \exists a' \in M,\ r' \in r \text{ s.t. } \textsc{OccludesPre}(o', a') \vee \textsc{OccludesManip}(a', o', r')\}$. Intuitively, this heuristic counts how many objects must be moved from state s. However, it is an underestimate of the actual remaining cost to the goal because it assumes each object only has to be moved once.

3. SAHS-BE-HCOUNT: SAHS with a Q-value function learned using the same planning experience data as ours and the same GNN, but with Bellman error as its objective function. It uses H_{count} as a heuristic function in the priority function.

Our algorithm, SAHS-LM-HCOUNT, uses a priorty function based on the heuristic H_{count} and the Q-function learned using the large-margin objective.

The learning algorithms have randomness from initialization of the neural network, and the planning algorithms have randomness from state computation and sampling of continuous parameters. We test each planning algorithm with 5 different random seeds, in 100 unseen problem instances for each problem and environment. For learning-based guidance algorithms, we train the Q-functions with 5 different random seeds. Therefore, for each of the learning algorithms, we perform 25 runs, 5 planning runs for each of 5 learning runs, on each problem instance.

For both box-moving and cupboard environments, we have a form of pick-and-place as our (single) operator. For the box-moving environment, we have two-arm pick-and-place; for the cupboard environment we have a one-arm pick-and-place. The pick operation is defined by two continuous parameter vectors: (1) a grasp, denoted with (d, h, χ), that specifies a depth, d, as a fraction of the size of object in the approach direction, height, h, as a fraction of object height, and angle in the approach direction, χ, respectively, and (2) the pick base pose, (x_r^o, y_r^o, ψ_r^o), which represents the robot base pose relative to the pose of an object o, whose pose in global frame is (x_o, y_o, ψ_o). For the two-arm case, the grasp parameters (d, h, χ) specify where the mid-point of the grippers of two arms should be; for the one-arm case the parameters specify the midpoint of the right-arm's gripper fingers. The place operation is specified simply by the robot base pose in the world reference frame, and we assume that the arms are fixed when the robot's base is moving.

An inverse kinematics solver is called to generate (collision-free) arm configurations for picking at the specified base pose and grasp parameters. In the box-moving environment, once we determine the pick-and-place continuous parameters, we plan the base-motion plans for both picking and placing the objects. In the cupboard environment, we omit motion planning and simply check collisions at the pick and place configurations.

	Avg time (1 obj)	Success rate	Avg time (4 objs)	Success rate
SAHS-LM-HCOUNT	$\mathbf{32.4 \pm 2.1}$	**0.96**	$\mathbf{78.0 \pm 6.1}$	**0.99**
SAHS-BE-HCOUNT	59.9 ± 4.9	0.94	165.9 ± 18.8	0.95
SAHS-HCOUNT	83.2 ± 6.6	**0.96**	289.5 ± 19.9	0.97
I-RSC	84.2 ± 9.9	0.82	317.7 ± 21.4	0.90

Table 1: Average time (in seconds), 95% confidence intervals, and success rates (with time limit) for placing boxes in the kitchen region. First two columns: placing 1 box (300s limit). Latter two columns: placing 4 boxes (1200s limit). Learning-based methods trained on 5000 examples.

Table 1 compares the average planning time and success rates for our benchmarks in the box-moving domain. We can see that the learning-based algorithms that use our representation and architecture outperform other methods, with our large-margin-based approach performing the best. The reason that SAHS-HCOUNT performs poorly is because, as a primarily state-based heuristic, it does not provide much guidance about which abstract edges to focus on. I-RSC performs reasonably well in the one object case, but poorly compared to other algorithms; this is partly because the algorithm makes the *monotonicity* assumption, which is that the problems can be solved by touching each object only once. An example nonmonotonic problem instance that can be solved by SAHS but not by I-RSC is provided in the supplementary materials. This indicates that the learned Q-function can guide nonmonotonic problem instances even though it was trained on experience from only monotonic problem instances.

	Avg time	Success rate
SAHS-LM-HCOUNT	$\mathbf{163.9 \pm 21.2}$	**0.90**
SAHS-BE-HCOUNT	220.8 ± 16.8	0.88
SAHS-HCOUNT	248.3 ± 31.2	0.86
RSC	317.9 ± 45.5	0.71

Table 2: Average time (in seconds), 95% confidence intervals, and success rates (with 1000s time limit) for moving a small object from the red cupboard to the box. Learning-based methods trained on 5000 examples from other domain.

We have found (see supplement) that the large-margin objective, which penalizes actions not seen in the dataset, performs better than the Bellman-error objective function. Even with 100 data points (about 30 planning episodes) SAHS-LM already outperforms RSC, while SAHS-BE-HCOUNT requires 1000 data points (about 250 planning episodes) to reach the performance level of RSC.

We now evaluate whether the same Q-function, without retraining, can be applied to an entirely different environment shown in Figure 1 (right). Here, the objective is to pack a single small object, using one-hand pick-and-place, into the green box. We assume the robot simply drops the object into the box, so as long as it samples a feasible placement in the region defined as the top of the box, it is considered to have achieved the goal. This problem is, in general, harder than the box-moving domain in the sense that it has more objects and tighter regions. Table 2 shows the results. As we can see, SAHS-LM-HCOUNT again outperforms other methods. RSC performs the worst: unlike in the previous domain, this environment has much less free space, which makes it much easier without the restriction to monotonic plans that is built into RSC.

Conclusion We proposed a relational representation and a GNN architecture which allow us to train a Q-value function to guide the search for abstract actions in GTAMP problems. We demonstrated the data efficiency of the large-margin loss function and showed that our representation affords generalization to hard problem instances and to substantially different environments.

Acknowledgement We gratefully acknowledge support from NSF grants 1523767 and 1723381; from AFOSR grant FA9550-17-1-0165; from ONR grant N00014-18-1-2847; from Honda Research; and from the MIT-Sensetime Alliance on AI. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] M. Stilman, J.-U. Schamburek, J. Kuffner, and T. Asfour. Manipulation planning among movable obstacles. *IEEE International Conference on Robotics and Automation*, 2007.

[2] A. Krontiris and K. E. Bekris. Dealing with difficult instances of object rearrangement. In *Robotics: Science and Systems*, 2015.

[3] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 2017.

[4] R. Chitnis, L. P. Kaelbling, and T. Lozano-Pérez. Learning quickly to plan quickly using modular meta-learning. *IEEE International Conference on Robotics and Automation*, 2019.

[5] B. Kim, L. P. Kaelbling, and T. Lozano-Pérez. Guiding search in continuous state-action spaces by learning an action sampler from off-target search experience. *AAAI Conference on Artificial Intelligence*, 2018.

[6] B. Kim, L. P. Kaelbling, and T. Lozano-Pérez. Adversarial actor-critic method for task and motion planning problems using planning experience. *AAAI Conference on Artificial Intelligence*, 2019.

[7] C. Guestrin, D. Koller, C. Gearhart, and N. Kanodia. Generalizing plans to new environments in relational MDPs. In *International Joint Conferences on Artificial Intelligence*, 2003.

[8] P. Tadepalli, R. Givan, and K. Driessens. Relational reinforcement learning: An overview. In *International Conference on Machine Learning*, 2004.

[9] S. W. Yoon, A. Fern, and R. Givan. Learning heuristic functions from relaxed plans. In *International Conference on Automated Planning and Scheduling*, 2006.

[10] M. Fink. Online learning of search heuristics. In *Artificial Intelligence and Statistics*, 2007.

[11] C. Domshlak, E. Karpas, and S. Markovitch. To max or not to max: Online learning for speeding up optimal planning. In *AAAI Conference on Artificial Intelligence*, 2010.

[12] C. R. Garrett, L. P. Kaelbling, and T. Lozano-Pérez. Learning to rank for synthesizing planning heuristics. In *International Joint Conference on Artificial Intelligence*, 2016.

[13] J. Pinto and A. Fern. Learning partial policies to speedup MDP tree search via reduction to I.I.D. learning. *Journal of Machine Learning Research*, 2017.

[14] R. Chitnis, D. Hadfield-Menell, A. Gupta, S. Srivastava, E. Groshev, C. Lin, and P. Abbeel. Guided search for task and motion plans using learned heuristics. In *IEEE International Conference on Robotics and Automation*, 2016.

[15] M. Bhardwaj, S. Choudhury, and S. Scherer. Learning heuristic search via imitation. In *Conference on Robot Learning*, 2017.

[16] A. Sperduti and A. Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 1997.

[17] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *IEEE International Joint Conference on Neural Networks*, 2005.

[18] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 2009.

[19] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[20] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

[21] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

[22] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. *International Conference on Machine Learning*, 2017.

[23] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations*, 2017.

[24] K. Hauser. The minimum constraint removal problem with three robotics applications. *The International Journal of Robotics Research*, 33(1), 2014.

[25] A. Farahmand. Action-gap phenomenon in reinforcement learning. In *Advances in Neural Information Processing Systems*, 2011.

Appendix for learning value functions with relational state representations for guiding task-and-motion planning

Beomjoon Kim **Luke Shimanuki**
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
{beomjoon,lukeshim}@mit.edu

1 Learning curve comparing Bellman-error and Large margin loss

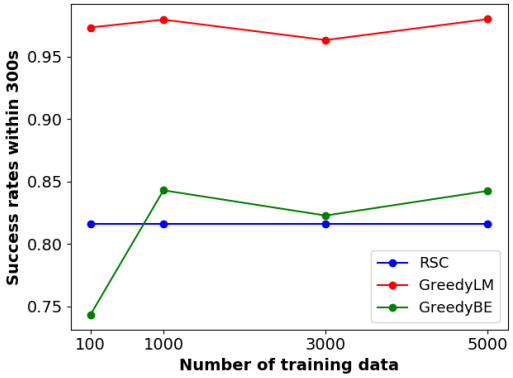


Figure 1: Learning curve of success rates for packing 1 box given 300s time limit in the box-moving domain

2 Details of GNN architecture

For all $f(\cdot, \theta_i)$, we use two fully-connected layers, each of which has 32 nodes. We use ReLU activation function for all nodes, except at the nodes of the last layer in $f(\cdot, \theta_4)$ and thatof $f(\cdot; \theta_5)$, which have the linear activation function. All layers have bias terms. We used Keras and tensorflow to implement the GNN. We perform two rounds of message passing in all of our experiments.

3 Predicate Evaluation and Caching

3.1 Predicate Implementations

The set of predicates we propose can apply to a wide range of geometric problem domains and operator classes. However, the manner of computing each predicate will vary between operator classes. We briefly describe the implementation details for each predicate in each domain we evaluated. We use the OpenRave inverse kinematics (IK) solver for the PR2 arms (given a sampled (x, y, θ) base pose for the robot). We also use a hand-built Probabilistic RoadMap (PRM) for motion planning between specified (x, y, θ) base poses, with fixed arm configurations.

3rd Conference on Robot Learning (CoRL 2019), Osaka, Japan.

3.1.1 Box-Moving Domain: Two-Arm Pick-and-Place

- PREFREE(b): Sample feasible robot base poses and inverse kinematic solutions for picking b, then call the motion planner to find a collision-free path to any sampled base pose. This predicate is true if a collision-free path can be found.
- MANIPFREE(b, r): Sample feasible placements for b in region r, as well as robot base poses and inverse kinematic solutions, then call the motion planner to find a collision-free path from any pick base pose to any sampled place base pose. This predicate is true if a collision-free path can be found.
- OCCLUDESPRE(b_1, b_2): Sample feasible robot base poses and inverse kinematic solutions for picking b_2, then call the motion planner to find a collision-free path to any sampled base pose. If no such path can be found, instead find a path that ignores collisions. Then this predicate is true if the selected path collides with b_1.
- OCCLUDESMANIP(b_1, b_2, r): Sample feasible placements for b_2 in region r, as well as robot base poses and inverse kinematic solutions, then call the motion planner to find a collision-free path from any pick base pose to any sampled place base pose. If no such path can be found, instead find a path that ignores collisions. Then this predicate is true if the selected path collides with b_1.

3.1.2 Cupboard Domain: One-Arm Pick-and-Place

- PREFREE(b): Sample feasiblerobot base poses and inverse kinematic solutions for picking b. This predicate is true if a collision-free arm solution can be found.
- MANIPFREE(b, r): Sample feasible placements for b in region r, as well as robot base poses and inverse kinematic solutions. This predicate is true if a collision-free solution can be found.
- OCCLUDESPRE(b_1, b_2): Sample feasible robot base poses and inverse kinematic solutions for picking b_2. If no collision-free solution can be found, instead find a solution that ignores collisions. Then this predicate is true if the selected arm configuration collides with b_1.
- OCCLUDESMANIP(b_1, b_2, r): Sample feasible placements for b_2 in region r, as well as robot base poses and inverse kinematic solutions. If no collision-free solution can be found, instead find a solution that ignores collisions. Then this predicate is true if the selected arm configuration collides with b_1.

3.2 Caching

Evaluating each predicate is usually a very expensive operation, but there is a lot of information that can be retained across different calls, or across different iterations within the same call, to the predicate evaluation function. Therefore, we use caching extensively to make the repeated computation of the predicates efficient. Although these techniques are tied to our specific implementation, the approach is quite general.

3.2.1 Probabilistic Roadmap for Motion Planning

A motion planner is called to sample the continuous operator parameters in the box-moving domain. It is also used to evaluate the predicates PREFREE, MANIPFREE, OCCLUDESPRE, and OCCLUDESMANIP. In our environment, the walls and other fixed objects remain constant across all problem instances, as only the movable objects have varying initial poses. Therefore, we pre-compute a finely-sampled probabilistic roadmap (PRM) that ignores movable objects but respects fixed objects. Later, when doing the graph search for a path, we check for collisions for motions on the edges of the PRM against the movable objects in the state. This leads to more efficient and less variable motion planning calls.

3.2.2 Cached Collisions and Paths

In a single state we make many motion planning calls. Performing collision checks between movable objects and robot configurations for each motion planning call can be quite expensive. So we cache which configurations in the PRM collide with each object in the current state, then reuse that information in future graph searches. We also retain collision-free paths that are reused in multiple predicate evaluations.

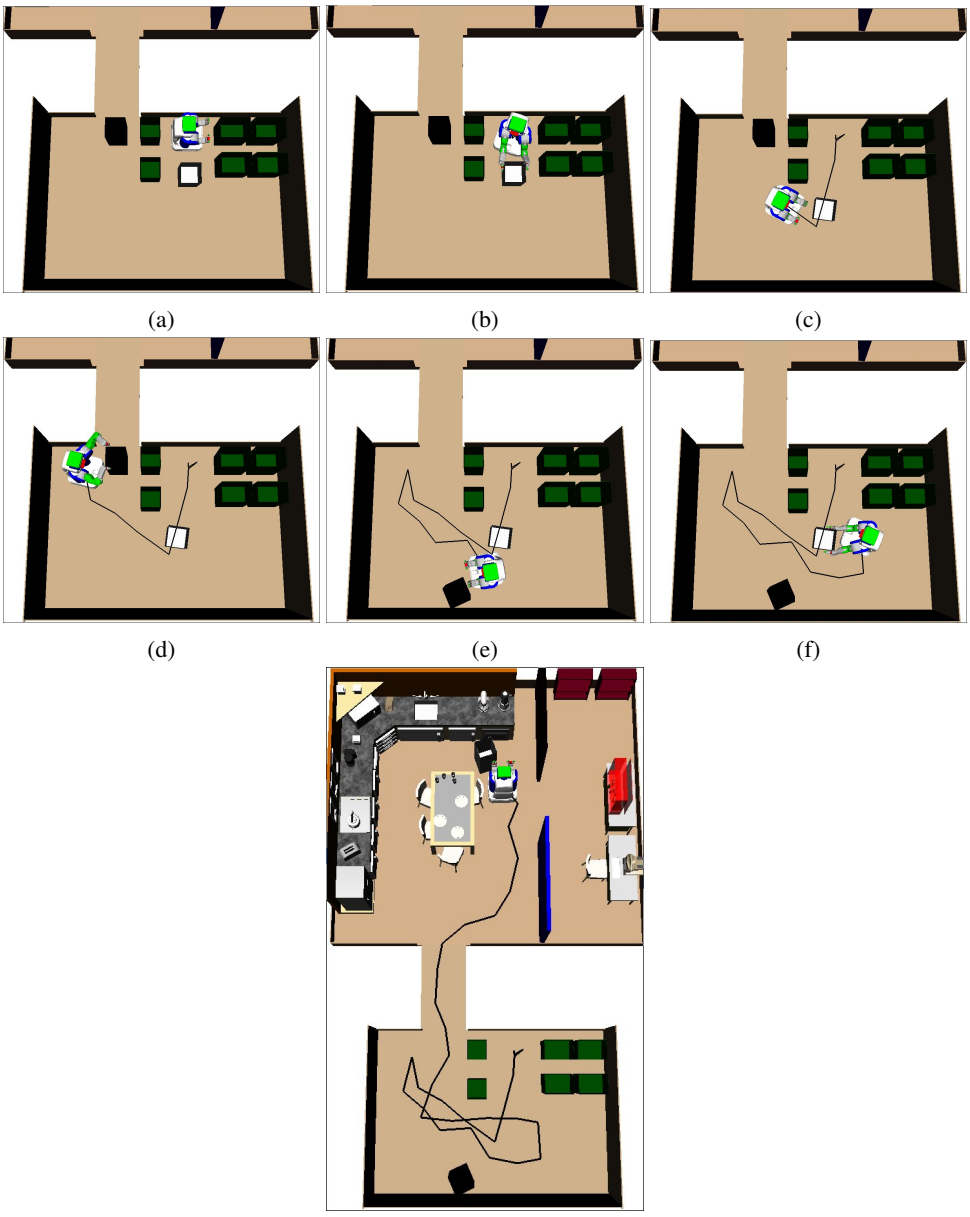
3.2.3 Inverse Kinematics Solutions

Unlike in the box-moving domain, in which collisions mostly constrain the space of feasible trajectories of the robot base, collisions in the cupboard domain heavily constrain the space of feasible arm configurations instead. Therefore, many inverse kinematic solutions are in collision, and so we must sample many configurations in order to find a feasible operator. Because inverse kinematic solving is a relatively expensive operation, this severely impacts the efficiency of planning and evaluating predicates. The workaround we use is to pre-compute a large number of inverse kinematic solutions for objects at a wide variety of poses relative to the robot base. Then at planning time we adapt the cached configuration by using relative transformations to make it fit with the actual object pose (for a pick operation) or the desired object placement (for a place operation). Many of these cached solutions will still be in collision, but avoiding the cost of finding the kinematic solution leads to significant speedups.

3.2.4 Predicate Evaluations

Finally, when an action is applied to a state, resulting in a new state, a lot of information can be passed down to improve the efficiency of evaluating the predicates for the new state. First, for any given action, many predicates will not change because moving a single object will leave most relationships between other objects the same, so we reuse the predicate value without recomputing it. Additionally, the set of PRM configurations in collision with each object only changes for the object that was moved, and so all other sets of collisions can be reused. In both domains, inverse kinematics solutions that are known to not collide with fixed objects can be retained for all objects except for the one that was moved. These configurations might still collide with movable objects, though.

4 Solving Nonmonotonic Problem Instances



(a) (b) (c)

(d) (e) (f)

(g)

Figure 2: An example of a problem instance that is nonmonotonic, in that the robot cannot move the goal object (white) to the goal region (the upper region) by only touching each object at most once. In this particular instance, the black box is blocking the corridor to the goal region, so the robot cannot move the white box directly to the goal region. However, the white box is blocking the robot from reaching the black box. Therefore, the robot must move the white box out of the way in order for it to move the black box, then it must touch the white box again to move it to the goal. Nonmonotonic problem instances such as this cannot be solved by Stillman's algorithm by design, but the greedy search planner with our learned Q function finds a plan in 30.4 seconds, averaged across 6 training and 20 planning seeds. This is interesting because it is able to learn to solve nonmonotonic problems even though it was trained on planning experience only from monotonic problems.