Policy-Guided Lazy Search with Feedback for Task and Motion Planning

Mohamed Khodeir[†] **Atharv Sonwane** **Florian Shkurti**[†] [*]

Abstract: PDDLStream solvers have recently emerged as viable solutions for Task and Motion Planning (TAMP) problems, extending PDDL to problems with continuous action spaces. Prior work has shown how PDDLStream problems can be reduced to a sequence of PDDL planning problems, which can then be solved using off-the-shelf planners. However, this approach can suffer from long runtimes. In this paper we propose `LAZY`, a solver for PDDLStream problems that maintains a single integrated search over action skeletons, which gets progressively more geometrically informed as samples of possible motions are lazily drawn during motion planning. We explore how learned models of goal-directed policies and current motion sampling data can be incorporated in `LAZY` to adaptively guide the task planner. We show that this leads to significant speed-ups in the search for a feasible solution evaluated over unseen test environments of varying numbers of objects, goals, and initial conditions. We evaluate our TAMP approach by comparing to existing solvers for PDDLStream problems on a range of simulated 7DoF rearrangement/manipulation problems.

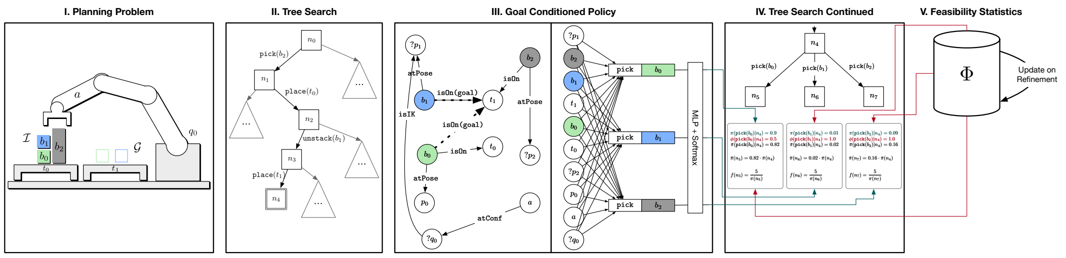
1 Introduction

Task and motion planning (TAMP) problems are challenging because they require reasoning about both discrete and continuous decisions that are interdependent. TAMP solvers typically decompose the problem by using a symbolic task planner that searches over discrete abstract actions, such as which object to interact with or what operations are applicable, and a motion planner which attempts to find the continuous parameters that ground those abstract actions, for instance grasp poses and robot configurations. The motion planner is responsible for informing the task planner when backtracking is necessary. Thus, this interplay between abstract task planning and low-level motion planning has a significant effect on the total runtime, or whether a solution is found at all.

In this work we provide a significantly improved PDDLStream[1] solver (`LAZY`) for task and motion planning problems, which learns to plan from past experience and current execution data. The motion planner of this solver provides feasibility updates to a priority/guidance function that is used to inform abstract action selection by the symbolic task planner. `LAZY` plans optimistically and lazily (deferring motion sampling until an action skeleton has been found), and maintains a single unified symbolic search tree (as opposed to solving a sequence of PDDL problems over a growing set of logical facts, as is done in [1]).

Our method additionally learns a goal-conditioned policy from past planning demonstrations. This policy affects a priority function, which guides the action skeleton search performed by the task planner towards promising abstract action sequences. We show how the predictions of this priority function can be updated online in response to failed samples in motion planning, allowing successive iterations of the task planner to focus the search on alternative action sequences. The result is a policy-guided bi-level search for TAMP problems, which improves online from experience and past data, and demonstrates impressive planning performance on unseen environments from a test distribution, while being trained with only a few hundred demonstrations. We empirically evaluate

[*†]Robot Vision and Learning Laboratory, University of Toronto Robotics Institute. `m.khodeir@mail.utoronto.ca, atharvs.twm@gmail.com, florian@cs.toronto.edu`


I. Planning Problem
II. Tree Search
III. Goal Conditioned Policy
IV. Tree Search Continued
V. Feasibility Statistics
Update on Refinement
MLP + Softmax
Φ
Figure 1: [I.] A 2D depiction of a planning problem. Three blocks are initially placed on the table on the left side (t_0). The blue and green blocks (b_0, b_1) must be unstacked and moved to the table on the right (t_1), however a tall grey block (b_2) obstructs any grasp. [II.] A snapshot of tree search where the node being expanded (n_4) corresponds to partial skeleton which first moves b_2 and then b_1 to t_1. The policy is now queried to determine which action to explore next. [III.] The state corresponding to node n_4 is encoded as a graph and passed to a Graph Attention Network (GAT) which produces a contextual embedding of each object. A second GAT produces an embedding of the applicable actions, and the result is passed to an output layer which computes a softmax. [IV.] The tree search continues with node priorities of n_4's children having been computed using the policy and empirical action feasibility estimates from the database Φ. [V.] When an action skeleton is found, and a refinement attempt fails, Φ is updated, leading to new priorities in subsequent tree search iterations.

our proposed method compared to existing approaches for sampling-based TAMP, such as [1], show significant (37%) improvement in the number of unseen problems solved.

2 Our Approach

2.1 Lazy Bi-Level Search

Our overall framework is a bi-level search, similar to prior work ([1], [2]). In every iteration, we search for an action skeleton $\vec{a}$ which is a sequence of discrete and high-level actions, with continuous parameters left as variables (for instance, grasp poses and placement poses). See Fig. 2 for an example. This outer search for an action skeleton is guided by a priority function f. We describe how f is defined in section 2.2 and elaborate on the details of skeleton search in section 2.3.

Once an action skeleton is found in the outer search, we perform the inner search for the assignments (groundings) of the continuous valued parameters in $\vec{a}$. We refer to this as *skeleton refinement*, and elaborate on it in section 2.4. The overall procedure terminates when refinement is successful, in which case a complete trajectory is returned. Otherwise, the result of the refinement is used to update the priority function f, and the next iteration begins, yielding a potentially different action skeleton. We refer to the process of incorporating the result of refinement into the priority function used by the outer search as *feedback* and detail our approach in section 2.5. The search fails to solve a given problem if the allotted planning time runs out before a trajectory is found. We provide pseudocode of this overall algorithm in the appendix.

2.2 Skeleton Search Routines and their Priority Functions

We build on ideas from Levin Tree Search (LevinTS) [3] as a way to incorporate a policy to guide the search while maintaining guarantees about completeness and search effort that relate to the quality of the policy. We assume that we are given a policy $\pi(a|s, \mathcal{G})$ which predicts a probability distribution over applicable actions (i.e. logical state transitions) conditioned on a logical state $s \in \mathcal{S}$ and goal $\mathcal{G} \subset \mathcal{S}$, where $\mathcal{S}$ is the set of all logical states. Given a node n in the search tree and its corresponding state and action sequence $s_0, ..., s_k$ $a_0, ..., a_{k-1}$, we use $\pi(a|s, \mathcal{G})$ to define the LevinTS priority function $f(n)$ based on $\pi(a|s, \mathcal{G})$ and the length of the sequence leading up to n, which we denote $d_0(n)$:

$$\pi(n) = \prod_{i=0}^{k-1} \pi(a_i|s_i, \mathcal{G}), f(n) = \frac{d_0(n)}{\pi(n)} \tag{1}$$

LevinTS prioritizes nodes n with low f value, namely nodes with high probability under Eqn. 1 and reachable by fewer actions than other leaf nodes in the search tree. We use behavior cloning to train the policy $\pi(a|s, \mathcal{G})$ from demonstrations.

2.3 Lazy Stream Instantiation in Skeleton Search

Our outer search for an action skeleton is 'lazy' in two respects. First, it is lazy in that it defers invoking any stream samplers until a full action skeleton has been found. In this respect, it is identical to that of the 'optimistic' variants of PDDLStream algorithms described in prior work [1]. Second, the search is 'lazy' in that streams are instantiated just-in-time, to support node expansion, as opposed to being eagerly instantiated in batch as in prior work. The main advantages of this is that (1) it allows a goal-seeking heuristic to guide the instantiation of streams and (2) it avoids the cost of the exhaustive search which is incurred by prior works when the instantiated streams are insufficient to support a plan.

In order to implement lazy stream instantiation, we modify the logical successor function used in the tree search to determine which streams need to be instantiated in order to produce the stream-certified preconditions of a logically applicable action. We use a simplified partial order planner to solve this problem for each applicable action. This can be done efficiently due to the restricted semantics of streams.

A by-product of performing this check, is that we construct a 'computation graph' (CG) for the parameters in our plan skeleton. This takes the form of a directed acyclic hypergraph where the root nodes are objects in $\mathcal{I}$, the hyper-edges correspond to streams, and the internal/leaf nodes are objects sampled from those streams. The CG for a given action skeleton defines a partial order over sampling operations for producing satisfying assignments of the skeleton's parameters. We maintain this structure at each node of the search tree. An example of a CG for a two step action skeleton is depicted in figure 2. Note that different skeletons may include objects with different identifiers which have the same computation graph. For example, any skeleton which includes an action that picks up an object is going to have a parameter corresponding to a grasp of that object. If we find that one such object has low feasibility (i.e. the sequence of streams that should produce it repeatedly fail), then this information should carry over to other plans which include 'similar' objects. Therefore, we define the concept of a CG *key*. If two objects share a CG key, this means that barring a renaming of variables, they have identical computation graphs.

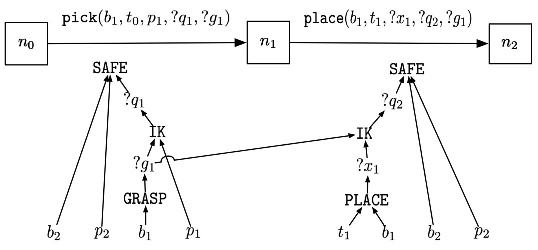


Figure 2: A plan skeleton to move an object b_1 from table t_0 to table t_1. There are four parameters which are unspecified: the grasp pose $?g_1$, its associated robot configuration $?q_1$, the placement pose $?x_1$ and its configuration $?q_2$.

2.4 Skeleton Refinement (Inner Search)

Skeleton refinement refers to the process of evaluating stream instances in the computation graph in order to produce assignments of a skeleton's continuous parameters. These sampling operations can fail if there are no feasible outputs conditioned on its inputs. For instance, as shown in Fig. 2, if we sample a particular grasp $?g_1$ for the pick action in the first step of the plan there may be no feasible inverse kinematics solution $?q_2$ for the subsequent placement action. In this work, we use a simple backtracking search which backtracks to the first action upon reaching a dead-end.

2.5 Incorporating Feedback

If skeleton refinement fails, this means that we were unable to find feasible assignments for one or more of the parameters of some action(s) in the skeleton. We would therefore like to modify the f value for failed actions so that the next iteration may avoid these in the search for a plan skeleton.

By maintaining statistics about the success or failure of stream instances in the computation graph of the skeleton, we can empirically estimate the probability of successfully sampling a feasible value for each parameter in the plan and identify bottleneck streams.

Since each action includes one or more parameters, we define the feasibility estimate for an action in a given state $\phi(a|s)$ as:

$$\phi(a|s) := \min_{\text{stream} \in a.streams} \frac{N_{success}^{\text{stream}} + 1}{N_{attempts}^{\text{stream}} + 1} \tag{2}$$

Note that before we have sampled a particular action's parameters, this definition leads to an estimate of $\phi(a|s) = 1$, meaning that we assume initially that all actions are feasible.

We incorporate these feasibility estimates into the f function in each iteration after a failed refinement. If during the course of sampling a candidate plan, we find that a_2 is infeasible, then we would like to **decrease** the policy's probability of taking that action in the next iteration. So, given an edge feasibility function $\phi(a|s) \to [0,1]$ we define $\bar{\pi}(a|s,\mathcal{G}) = \frac{\pi(a|s,\mathcal{G})\phi(a|s)}{\sum_{a'} \pi(a'|s,\mathcal{G})\phi(a'|s)}$.We use $\bar{\pi}(a|s,\mathcal{G})$ to define $\bar{\pi}(n)$ in the same way as described in equation 1.

Note that prior to obtaining empirical estimates for ϕ, we have $\bar{\pi} = \pi$. Furthermore, actions which are found to be infeasible are deprioritized online in subsequent iterations, allowing the task planner to focus on more promising action sequences.

3 Experimental Results

We use behavior cloning to train a single goal-conditioned policy $\pi(a|s,\mathcal{G})$ from a set of demonstrations collected by a TAMP solver. This training set consists of 400 problems of 4 different types. We test on a set of 500 held out test problems involving more objects and longer solutions.

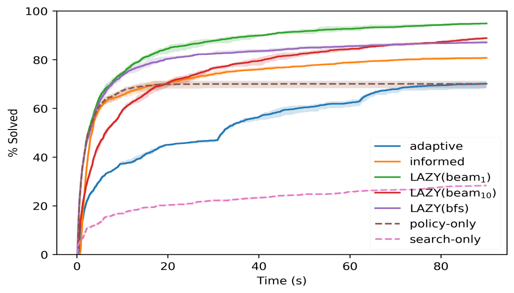


Figure 3: Percentage of problems solved as a function of planning time.

We evaluate 3 variants of policy-guided search from our framework. The first two (i.e. LAZY(beam$_1$), and LAZY(beam$_{10}$)) are instances of beam search with beam widths of $W = 1$ and $W = 10$ respectively. The third (i.e. LAZY(bfs)), is an instance of best-first-search. All of these variants use the LevinTS priority function from equation 1. In figure 3, we compare these to INFORMED [4], a prior work which uses learned models to prioritize the inclusion of stream instances according to their predicted relevance to the planning problem, and ADAPTIVE [1] which does not use learning. We find that LAZY(beam$_1$) is consistently the highest performer, solving an average of 96% of test problems within the allotted time.

We also evaluate two ablations of our method, shown in dotted lines in figure 3. The first, 'policy-only' simply uses the learned policy greedily to find a single plan skeleton which it tries to refine for the remainder of the time. The second, 'search-only' uses the lazy search with feedback framework without a guidance policy. The results show that, although the policy is effective at guiding search, it is not sufficiently good as to do away with search altogether, and benefits greatly from the overarching framework. Similarly, the relatively poor performance of 'search-only' demonstrates that both components contribute significantly to the overall success of our proposed approach.

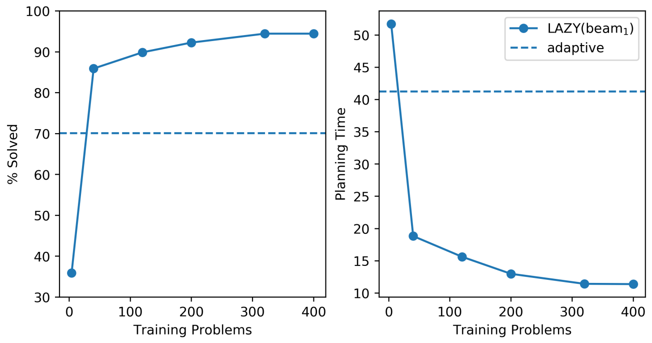


Figure 4: Left: Percentage of test problems solved by LAZY(beam$_1$) as a function of the training set size. Right: Average planning time as a function of training set size.

In order to shed light on the effect of training set size on the performance of the planner, we trained policies on increasing subsets of the full training set, and evaluated their performance on the test set. We report the percentage of problems solved within the 90 second timeout, as well as the average planning time as a function of the training set size. We find that LAZY(beam$_1$) outperforms baselines with only 50 training examples, and continues to improve on both metrics as more training data is used.

4 Conclusion

In this work, we proposed bi-level lazy search guided by learned goal-conditioned policies as a method for solving TAMP problems expressed using the PDDLStream formalism. We evaluated this approach experimentally against existing solvers, including prior work which uses learned models, and demonstrated significant improvements in planning times and solve rates across a range of unseen manipulation problems using a 7DoF robot arm.

References

[1] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling. Pddlstream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning, 2020.

[2] N. T. Dantam, Z. K. Kingston, S. Chaudhuri, and L. E. Kavraki. An incremental constraint-based framework for task and motion planning. *The International Journal of Robotics Research*, 37 (10):1134–1151, 2018. doi:10.1177/0278364918761570. URL `https://doi.org/10.1177/0278364918761570`.

[3] L. Orseau, L. Lelis, T. Lattimore, and T. Weber. Single-agent policy tree search with guarantees. *Advances in Neural Information Processing Systems*, 31, 2018.

[4] M. Khodeir, B. Agro, and F. Shkurti. Learning to search in task and motion planning with streams. *arXiv preprint arXiv:2111.13144*, 2021.

[5] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. Dahl, A. Vaswani, K. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu. Relational inductive biases, deep learning, and graph networks, 2018.

[6] T. Silver, R. Chitnis, A. Curtis, J. Tenenbaum, T. Lozano-Perez, and L. P. Kaelbling. Planning with learned object importance in large problem instances using graph neural networks, 2020.

[7] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=rJXMpikCZ`.

[8] S. Brody, U. Alon, and E. Yahav. How attentive are graph attention networks? In *International Conference on Learning Representations*, 2022. URL `https://openreview.net/forum?id=F72ximsx7C1`.

A Pseudocode

Algorithm 1: Lazy Bi-Level Search

```
def LAZY($n_0$, $\mathcal{G}$, $search$, $f$)
  $\Phi = \emptyset$
  while not timed out
    # $n$ is a descendent of $n_0$
    # $\vec{a}$ is the corresponding action sequence
    $\vec{a}, n := search(n_0, \mathcal{G}, f)$
    if $n$ = null
        break
    # maintain fail/success counts in $\Phi$
    $\vec{\theta}$ := refine($\vec{a}, N_{max}, \Phi$)
    if $\vec{\theta} \neq null$
        # actions and their grounded parameters
        return $\vec{a}(\vec{\theta})$
    # some step in the plan failed
    # update priority function $f$
    use $\Phi$ to update $f$
  return null # failure due to timeout
```

Algorithm 2: Skeleton Refinement

```
def refine($\vec{a}, N_{max}, \Phi$)
  for d := 0 to $N_{max}$
    $\vec{\theta} := \emptyset$
    for $a \in \vec{a}$
        for stream $\in$ a.streams
            o = next(stream)
            $\Phi$[stream].attempts++
            if o = null
                break # fail
            $\Phi$[stream].success++
            $\vec{\theta} := \vec{\theta} \cup \{o\}$
        if a failed
            if a never succeeded before # deadend
                break
            else
                change $\vec{\theta}$ to previous success
    return null if failed else $\vec{\theta}$
```

B Architecture and Training of the Skeleton Search Policy

As described in section 2.2, in order to guide the skeleton search (using the LevinTS priority function), we require a policy $\pi(a|s, \mathcal{G})$ which assigns a probability distribution over applicable actions in a given state. One challenge here is that, since we are performing a search in the space of plan skeletons, we only have access to the low-level state in the initial scene. This is because the actions (i.e. logical transitions) that we consider during our skeleton search have parameters (e.g. motions, poses, etc) which are left unspecified. For instance, a plan skeleton which places an object on the table will not specify the precise grasp used, or the precise final pose of the object on the table. We would like to learn a policy $\pi_\theta(a|\hat{s}, \mathcal{G})$ where $\hat{s} = \langle \mathcal{I}, \vec{a} \rangle$ describes the low-level initial state, and the logical partial skeleton, and $\mathcal{G}$ describes the set of desired facts in the goal.

B.1 State and Goal Representation

In this work, we consider policies parametrized by Graph Neural Networks [5]. An illustration of the end-to-end architecture is shown in figure 1. We encode a state s and goal $\mathcal{G}$ as a graph, with nodes representing objects (e.g. table2, block1, robot) and edges between them representing facts which hold (e.g. block1 is on table2) in s or $\mathcal{G}$, following prior work [6, 4]. Node features encode the type of object, the precise 3D pose (if unchanged from $\mathcal{I}$), and size of the object. We use Graph Attention Networks (GAT) [7, 8] to produce contextual embeddings for each of the objects.

B.2 Action Representation

The set of applicable actions $\mathcal{A}(s)$ at a given state comprise the domain of the probability distribution which should be predicted by the policy. Each action consists of the name of an operator (e.g. pick/place/stack/unstack) encoded using a 1-hot vector of fixed size, as well as a tuple of discrete parameters, whose encodings are obtained from the final layer of the GAT. We employ a second attention network which allows each of these actions to attend to every object in the state, and produce an embedding which is then passed to a simple multilayer perceptron and softmax layer to produce the final probability distribution over actions.

B.3 Partial Plan Representation

We experimented with using a recurrent neural network (RNN) on the sequence of states resulting from the partial plan $\vec{a}$ to obtain a history-dependent encoding. However, we found that simply discarding the partial plan, and using only the graph corresponding to the final logical state of the partial plan led to similar prediction accuracy and superior planning performance. We hypothesize that history features were not required in our tasks, but may be important under different circumstances.

B.4 Training and Data Collection

We use behavior cloning to train the policy from demonstrations. We generate these by using our algorithm with the A* priority function on the set of training problems $\{\mathcal{I}^{(i)}, \mathcal{G}^{(i)}\}_{i=1}^{N}$.

Note that the returned action sequences $\vec{a}(\vec{\theta})^{(i)}$ will have all of their continuous parameters fully specified. In order to train the policy for skeleton search, we extract the high level actions $a_{1:T^{(i)}}^{(i)}$ from the returned trajectory, and use the known high level transition function to extract the sequence of high level states $s_{0:T^{(i-1)}}^{(i)}$.

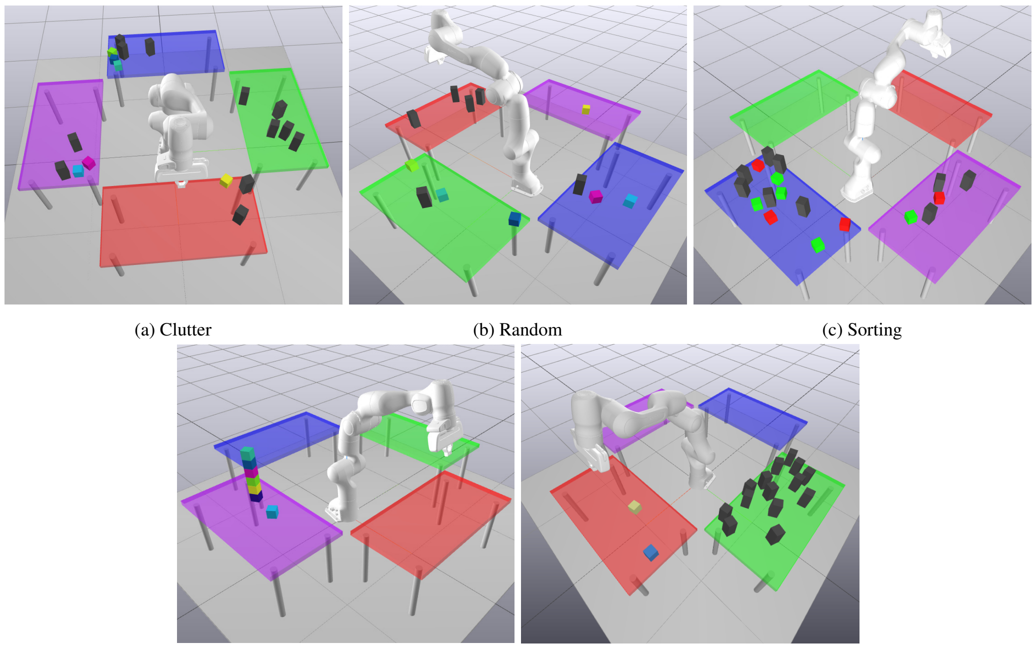
We then construct a behavior cloning dataset consisting of goal, state and action tuples $\{\langle \mathcal{G}^{(i)}, s_{j-1}^{(i)}, a_j \rangle\}_{i=1,j=1}^{i=N,j=T^{(i)}}$ and train our models to minimize the cross-entropy loss between the demonstration and predictions.

C Experimental Details

C.1 Problem Types

Evaluation of LAZY was conducted across five problem types involving a 7DoF robot arm. Problems are divided into five categories which share a domain definition, but present different challenges to the planner. Example scenes are shown in figure 5. There are two types of blocks in these problems (not distinguished logically): blocks (shorter) and blockers (taller). The robot arm is surrounded by four tables.

In Stacking, blocks are arranged randomly in each scene, and the goal is to assemble them into specific towers. In Sorting, the goal is to move colored blocks to the table with the corresponding color. Blockers may need to be moved if they obstruct a plan, but must be returned to their original tables. Test problems involved up to 10 blocks and 10 blockers. In Random, blocks need to be stacked or rearranged, and blockers may obstruct actions. Clutter problems are similar to Random,


(a) Clutter (b) Random (c) Sorting

(d) Stacking (e) Distractors

Figure 5: Typical example of the evaluation tasks in each environment.

but contain twice as many blockers, and initial positions are sampled using ordered Poisson-Disc Sampling so that there is a higher chance of obstructions. Finally, in Distractors, blocks need to be stacked or rearranged in the presence of "distractor" objects which are placed on a separate table. Unlike the blockers in other problem types, distractors do not appear in the goal, and never need to be interacted with. This problem type tests the planner's ability to ignore irrelevant objects.

For each problem type, we randomly generate 100 instances which are used to train the model and 100 held-out test instances which we use for evaluation. We do not train the model on any Distractors problems, but instead reserve these just for testing. *The training instances are drawn from distributions with few objects/goals so that the baseline is able to solve the majority of them within the timeout. We sample more challenging instances from a different distribution for testing to evaluate the model's ability to generalize to harder problems with more objects than those seen during training.* Initial placement of objects and their goal states are randomly generated in each problem, so that even problems of the same type with the same number of objects will have different solutions.

All experiments are conducted using 2 cores of an Intel Broadwell processor with an 8GB memory limit. All methods use a 90 second planning timeout, so we report the proportion of problems solved within the timeout, as well as average planning times for solved instances.