2023 IEEE International Conference on Robotics and Automation (ICRA 2023)
May 29 - June 2, 2023. London, UK

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

Mohamed Khodeir[1] Atharv Sonwane[2] Ruthrash Hari[3] Florian Shkurti[1]

***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. Code can be found at `https://rvl.cs.toronto.edu/learning-based-tamp`.**

I. 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 informs the task planner when backtracking is necessary. Thus, the interplay between abstract task planning and low-level motion planning has a significant effect on both runtime and percentage of problems solved.

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

[1]Robot Vision and Learning Lab, University of Toronto Robotics Institute.
[2]Dept. of Computer Science, University of BITS Pilani
[3]Dept. of Mathematical and Computational Sciences, University of Toronto
{m.khodeir, ruthrath.hari}@mail.utoronto.ca, atharvs.twm@gmail.com, florian@cs.toronto.edu

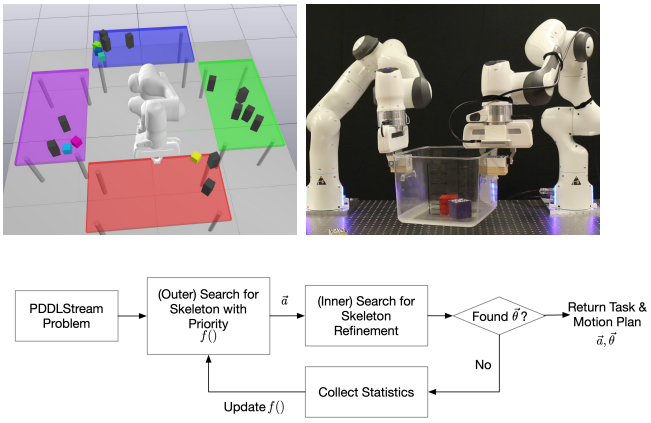


Fig. 1: Top Left: Simulated evaluation tasks in Clutter environment. Top Right: Real-world manipulation problems using two 7DoF robot arms. Bottom: A flowchart illustrating the high level components of our approach.

A core component of our method is a goal-conditioned policy over high-level actions, which we learn using behaviour cloning on past planning demonstrations. This policy is treated as a priority function, which guides the action skeleton search performed by the task planner towards promising abstract action sequences. While this can often eliminate the need for backtracking altogether, the policy may still predict geometrically infeasible actions in more challenging TAMP problems. We therefore 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 more feasible 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.

Our main contributions are: (1) A lazy search framework for PDDLStream problems, which maintains a single search tree over symbolic plan skeletons. (2) A method for incorporating a learned policy over symbolic actions into sampling-based bi-level search, and efficiently updating it online using feedback from motion planning. (3) A concrete parametrization of this goal conditioned policy as a Graph Attention Network (GAT) which incorporates both high-level and low-level state. We empirically evaluate our proposed method compared to existing approaches for sampling-based TAMP, such as [1] and show significant (37%) improvement in the number of unseen problems solved within the allotted planning time.

II. Background: PDDL and PDDLStream

We adopt PDDLStream [1] as the formalism for expressing TAMP problems. A PDDLStream domain $(\mathcal{P}, \mathcal{A}, \Psi)$ is

defined by predicates $\mathcal{P}$, actions $\mathcal{A}$, and streams Ψ.

At a high level, predicates are boolean valued n-ary functions which indicate the presence of particular relations among their variables. For instance, the predicate `isOn` may indicate that the object in its first argument is on top of the object in its second. When a predicate is applied to specific objects (e.g. `isOn(A, B)`), we refer to it as a "fact".

Actions define the legal state transitions in the planning problem. They are defined by a set of parameters, a set of preconditions which define facts on those parameters which must hold in order for the action to be applicable, and effects that determine which facts about the parameters are added or removed following the application of the action.

The set of streams, Ψ, distinguishes a PDDLStream domain from traditional PDDL. Streams are conditional generators which yield objects that satisfy specific constraints conditioned on their inputs. Formally, a stream, s, is defined by input and output parameters $\bar{x}$, $\bar{o}$, a set of facts $domain(s)$, and a set of facts $certified(s)$. $domain(s)$ is the set of facts that must evaluate to true for an input tuple $\bar{x}$ to be valid. This ensures the correct types of objects (e.g., configurations, poses etc.) are provided to the generators. $certified(s)$ are facts about $\bar{x}$ and $\bar{o}$ that will be true of any outputs $\bar{o}$ that the stream generators produce. Streams can be applied recursively to generate a potentially infinite set of objects and their associated facts, starting from those in $\mathcal{I}$. They can also be thought of as declaratively specifying constraints between their inputs and outputs. Finally, each stream comes with a black-box procedure which, given input values $\bar{x}$, produces samples $\bar{o}$ which satisfy those constraints. We use the term stream *evaluation* to refer to the act of querying this sampler.

The PDDLStream domain $(\mathcal{P}, \mathcal{A}, \Psi)$ defines a language in which to pose specific problems. An instance of a planning problem in this domain is defined by specifying the initial state $\mathcal{I}$ which is simply a set of facts using predicates $\mathcal{P}$ that describe the initial scene, and the goal $\mathcal{G}$. $\mathcal{I}$ and $\mathcal{G}$ implicitly define a set of initial objects over which facts in those sets are stated. A solution to a problem instance consists of a sequence of action instances which result in a state in which $\mathcal{G}$ is satisfied. Note that many of the parameters in a solution may need to be produced using the streams and initial objects.

Predicates in classical PDDL problems can be classified as either "static" or "fluent" depending on whether they appear in the effects of any action. Static predicates are used to define types (e.g. `isTable(x)`) or immutable relations between objects (e.g. `isSmaller(x, y)`). Fluent predicates, on the other hand, are those which can be changed by actions (e.g. `isOn(x, y)`). By definition, streams are only allowed to certify "static" predicates (e.g. `isGraspPose(x)`). Therefore, in PDDLStream problems, we can further categorize static predicates based on whether they are produced by streams or are simply given in the initial conditions $\mathcal{I}$. We call the former stream-certified preconditions.

We use the notation $\vec{a}$ to refer to an "action skeleton", which is a sequence of discrete, high-level action instances with continuous parameters left as variables (for instance, grasp poses and placement poses). See Fig. 3 for an example of a two-step action skeleton. We denote a specific assignment/grounding of continuous parameters as $\vec{\theta}$, and refer to the grounded plan as $\vec{a}(\vec{\theta})$. Similarly, we use a, θ and $a(\theta)$ to refer to individual actions and their grounding.

III. Our Approach

A. Lazy Bi-Level Search

Our overall framework is a bi-level search, similar to prior work on task and motion planning ([1], [2]). In every iteration, we search for an action skeleton $\vec{a}$. This outer search for an action skeleton is guided by a priority function f, which assigns a lower value to more desirable actions. We describe possible choices for how f is defined in section III-B and elaborate on the details of skeleton search in section III-C.

Once an action skeleton is found in the outer search, we perform the inner search for grounding its continuous parameters $\vec{\theta}$. We refer to this as *skeleton refinement*, and elaborate on it in section III-D. The overall procedure terminates when refinement is successful, in which case a complete trajectory is returned. Otherwise, the result of the previous 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 a number of possible implementations in section III-E. The search fails to solve a given problem if the allotted planning time runs out before a trajectory is found. This overall framework is summarized in Algorithm 1 and illustrated in Figure 1.

B. Skeleton Search Routines and their Priority Functions

There are many possible choices for the skeleton `search` routine and its associated priority function f leading to algorithms with different characteristics. In this work, we consider two implementations of `search`: the first is a simple best-first search and the second is a beam search. Intuitively, decreasing the value of the beam width parameter in beam search allows us to create greedier search algorithms at the cost of potentially pruning out solution branches.

We also consider two implementations of f: first, the familiar A* priority function ($f(n) = g(n) + h(n)$), which we use to incorporate off-the-shelf domain-agnostic heuristics from prior work [3]. Note that this option allows our algorithm to work well without a learned policy, using existing domain-agnostic search heuristics in place of h. We make use of this for data collection, and as a baseline in evaluation.

Second, we build on ideas from Levin Tree Search (LevinTS) [4] as a way to incorporate a policy to guide the search while maintaining guarantees about completeness and search effort of the symbolic planner 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 discrete 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.

We distinguish between a state in the search space and a node in the search tree by using the symbol s to denote the

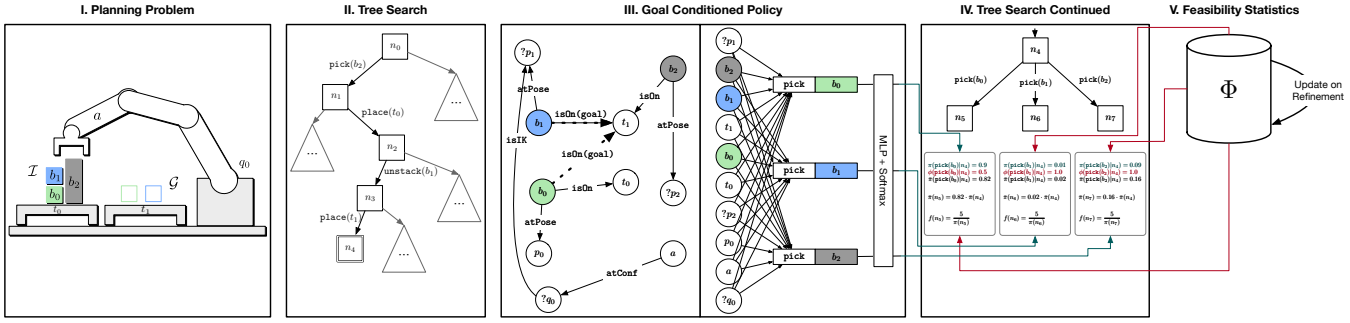


Fig. 2: [I] A 2D depiction of a planning problem. Three blocks are initially placed on the table on the left side (t_0). The blue and green blocks (b_0, b_1) must be unstacked and moved to the table on the right (t_1), however a tall grey block (b_2) obstructs any grasp. [II] A snapshot of tree search where the node being expanded (n_4) corresponds to a 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 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.

Algorithm 1: Lazy Bi-Level Search

```
def LAZY(n_0, G, search, f)
  Φ = ∅ # feasibility statistics
  while not timed out
    # a⃗ is an action skeleton that achieves G
    a⃗ := search(n_0, G, f)
    if a⃗ = null
        break
    # maintain fail/success counts in Φ
    θ⃗ := refine(a⃗, N_max, Φ)
    if θ⃗ ≠ null
        # actions and their grounded parameters
        return a⃗(θ⃗)
    # some step in the plan failed
    # update priority function f
    use Φ to update f
  return null # failure due to timeout
```

former and n to denote the latter. A node corresponds to a specific sequence of actions starting from the initial state $\mathcal{I}$. We use n_0 to refer to the root node of the search tree, which is the empty path starting from the initial state $\mathcal{I}$. Given a node n 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

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

The LevinTS priority function $f(n)$ depends on $\pi(n)$ and the length of the sequence leading up to n, which we denote $d_0(n)$, and is defined by:

$$f(n) = \frac{d_0(n)}{\pi(n)} \tag{2}$$

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.

C. 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,

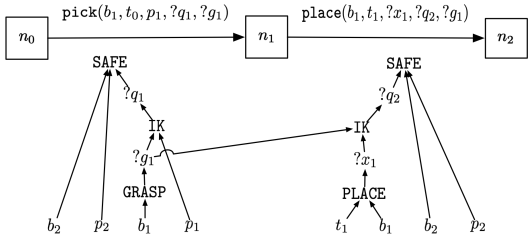


Fig. 3: A plan skeleton to move an object b_1 from table t_0 to table t_1. There are four parameters which are unspecified: the grasp pose $?g_1$, its associated robot configuration $?q_1$, the placement pose $?x_1$ and its configuration $?q_2$.

the outer 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 doing this are 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 eagerly instantiated streams are insufficient.

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. To do this, we need to check whether all stream-certified preconditions could be produced by a combination of the set of streams Ψ and the objects in the current state. We implement this check by casting it as a planning problem, where each stream defines a corresponding action with domain conditions playing the role of action preconditions, and certified conditions playing the role of action effects. We then solve for a sequence of streams that convert the initial state into a state where the desired stream-certified conditions hold. We use a simplified partial order planner to solve this problem for each applicable action.

A byproduct 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.

Algorithm 2: Skeleton Refinement

```
def refine(a⃗, N_max, Φ)
  for d := 0 to N_max
    θ⃗ := ∅
    for a ∈ a⃗
        for stream ∈ a.streams
            ō = next(stream)
            Φ[stream].attempts++
            if ō = null
                break # stream failed
            Φ[stream].success++
            θ⃗ := θ⃗ ∪ {ō}
        if all streams successful
            record θ⃗ as partial grounding of a
        elif any partial grounding θ⃗' of a exists
            θ⃗ := θ⃗'
        else break # deadend
    return θ⃗ if all actions grounded else null
```

We maintain this structure at each node of the search tree. We refer to the stream instances that comprise the CG of a high level action a as $a.streams$. An example of a CG for a two-step action skeleton is depicted in Figure 3.

D. 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 Figure 3, 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. Therefore, we have to be able to backtrack to reconsider the choice of grasp. This is common in sampling-based TAMP approaches, and there are many possible strategies that may be used. In this work, we use a simple strategy which backtracks to the first action upon reaching a dead-end. See Algorithm 2. It can be shown that this strategy is probabilistically complete if the streams produce samples with replacement.

E. 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 of Algorithm 1 may avoid them. 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 bottlenecks.

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

$$\phi(a|s) := \min_{\text{stream} \in a.streams} \frac{N^{\text{stream}}_{success} + 1}{N^{\text{stream}}_{attempts} + 1} \tag{3}$$

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 feasibility estimates into the f function in each iteration after a failed refinement. When f is defined as in A*, these estimates replace the unit cost associated to each action in the cost-to-come $g(n)$ - we detail this in section III-E.2. Similarly, we describe how these feasibility estimates are incorporated into the policy when using the LevinTS implementation of f in III-E.3.

1) Computation Graph Keys: As described in section III-C, each node in our search tree maintains a computation graph that defines the sequence of streams which produce each of the parameters/objects in the plan skeleton. 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 parameter 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" parameters. 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.

2) Feedback in A Priority:* When using the A* priority function for f, we define the cost of an action in our plan as $c(a|s) = \frac{1}{\phi(a|s)}$. This means that in the first iteration of planning, when the feasibility of actions is optimistic, the planner uses unit costs for all actions. Similarly, all actions whose computation graphs have never been encountered in refinement will have unit cost. On the other hand, actions whose parameters have failed to be refined will have their costs increased, and thus be deprioritized. This is akin to a relaxation of the binary edge evaluation in [5].

3) Feedback in LevinTS Priority: 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 of algorithm 1. So, given an edge feasibility function $\phi(a|s) \rightarrow [0, 1]$ 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 $\bar{\pi} = \pi$. Actions which are found to be infeasible are deprioritized in subsequent iterations.

F. Architecture and Training of the Skeleton Search Policy

As described in section III-B, in order to guide the skeleton search (using the LevinTS priority function), we require a policy $\pi(a|s, \mathcal{G})$ that 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 optimistically 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 $\pi(a|\hat{s}, \mathcal{G})$ 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.

1) State and Goal Representation: In this work, we consider policies parametrized by Graph Neural Networks [6]. An illustration of the end-to-end architecture is shown in Figure 2. 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 [7], [8]. 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) [9], [10] to produce contextual embeddings for each of the objects.

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.

3) Training and Data Collection: We use behavior cloning to train the policy from demonstrations on the set of training problems $\{\mathcal{I}^{(i)}, \mathcal{G}^{(i)}\}_{i=1}^{N}$. We generate these using Algorithm 1 with the A* priority function described in III-B and III-E.2.

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 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.

IV. Experimental Results

Our experiments are designed to shed light on the following research questions: (Q1) How well does LAZY perform when used with off-the-shelf domain-agnostic search heuristic? (Q2) How effective is the learned policy at guiding the skeleton search? (Q3) Which of the policy-guided search variants described in III best incorporate the learned policy?

A. 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 1. There are two types of blocks (not distinguished logically): blocks (shorter) and blockers (taller).

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, 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 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 placements and goals are randomly generated in each problem, so that even problems of the same type with the same number of objects will have different solutions.

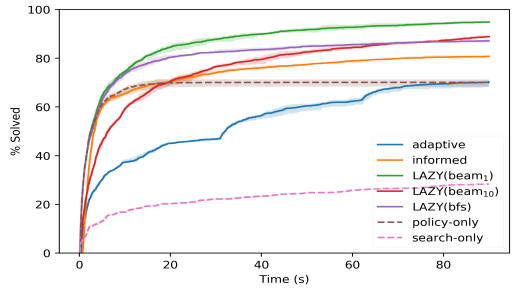
All experiments are conducted using 2 cores of an Intel Broadwell processor with an 8GB memory limit. All methods use a 90 second planning timeout, so we report the proportion of problems solved within the timeout, as well as average planning times for solved instances.

B. Results and Discussion

Problem	adaptive	LAZY(h_{add})
Random	72.00 ± 1.00	**91.80 ± 1.10**
Clutter	55.40 ± 1.52	**61.00 ± 1.41**
Stacking	61.40 ± 0.89	**88.00 ± 2.35**
Sorting	**77.20 ± 4.15**	68.00 ± 1.87
Distractors	86.20 ± 1.10	**99.80 ± 0.45**

TABLE I

To address (Q1), we evaluate LAZY(h_{add}) which uses the popular domain-agnostic heuristic "h_{add}" [3] using the A* priority function. This also establishes a baseline with which to compare the learned policy guided version of LAZY for (Q2). In the table on the left, we compare LAZY(h_{add}) to adaptive [1] in terms of the percentage of test problems solved during the allotted time. We find that across 4 out of 5 problem types, LAZY(h_{add}) solves significantly more problems within the allotted planning time. Adaptive only outperforms LAZY(h_{add}) on sorting problems. We found that this to be the result of increased node expansion time due to the larger number of objects/goals in those problems. As adaptive relies on the efficient implementation of FastDownward, it is able to handle this more effectively. We attribute the improvement on the remaining 4 problem



	Random	Clutter	Stacking	Sorting	Distractors
informed	89.8 ± 0.84	64.0 ± 1.00	73.0 ± 1.00	59.0 ± 1.82	**99.8** ± 0.45
LAZY(beam$_1$)	**99.0** ± 0.00	**90.8** ± 1.10	**91.6** ± 1.82	**97.2** ± 0.84	**100** ± 0.00
LAZY(beam$_{10}$)	**97.8** ± 0.45	82.0 ± 1.41	**93.6** ± 0.55	78.4 ± 3.36	96.6 ± 1.14
LAZY(bfs)	**98.0** ± 0.00	79.2 ± 1.64	**93.6** ± 0.55	68.8 ± 1.30	**100** ± 0.00
policy-only	83.8 ± 1.92	43.4 ± 0.55	76.2 ± 2.39	51.4 ± 0.89	98.6 ± 2.19
search-only	29.0 ± 0.00	24.2 ± 0.084	45.0 ± 0.00	12.0 ± 0.00	32.4 ± 0.55

Fig. 4: Figure shows solve rate as a function of planning time. Table reports percentage of problems solved within 90 second timeout. All variants of LAZY use the LevinTS priority function with the learned policy. We report the mean and standard deviation across 5 random seeds for each method.

types to the feedback process described in section $III-E.2$ enabling a more efficient search for a feasible plan skeleton.

To address (Q2/Q3) 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 2.

In Figure 4, we compare these variants of our approach to INFORMED [8], a prior work which uses learned models to prioritize the inclusion of stream instances according to their predicted relevance to the planning problem. While all of these methods outperform both non-learning baselines from table I, we find that LAZY(beam$_1$) is consistently the highest performer, solving an average of 96% of test problems in the allotted time. This suggests that the learned policy is effective at guiding search. However, in general, we expect that the answer to (Q3) will depend on the quality of the policy.

We also report the performance of two ablations of our method. 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 LAZY(beam$_1$).

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

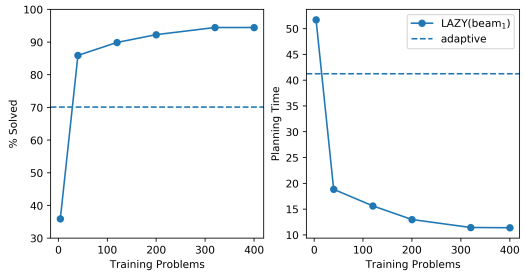


Fig. 5: Left: Percentage of test problems solved by LAZY(beam$_1$) as a function of the training set size. Right: Average planning time as a function of training set size.

metrics as more training data is used.

V. Related Work

Integrated task and motion planning. There is a vast literature on the problem of integrating the geometric reasoning required by motion planning with the symbolic reasoning that is necessary for planning to achieve abstract goals; see [11] for a detailed taxonomy. Our work builds on the PDDLStream [1] formalism, which we introduced in detail in the background section. Several algorithms for PDDLStream problems have since been proposed, including [12] which uses Monte-Carlo Tree Search to efficiently search for a low-cost solution. PDDLStream has also been used to facilitate belief-space planning in partially observed environments [13].

There is a long history of prior research, including [14], [15] combining symbolic planners with complete geometric planners. The need for selecting correct hierarchical abstractions for symbolic planning and favoring feasibility and real-time results over optimality was emphasized in [16]. Logic Geometric Programming combined symbolic planning and trajectory optimization [17], [18], even for dynamic physical motions involving tool use, while [19] integrated sampling procedures with SAT solvers. The idea of incorporating refinement failures to bias symbolic search away from infeasible actions was explored in [19], [20], [21].

Learning for TAMP. Motivated by the success of learning in the context of robotics, recent work has sought to combine the ability of TAMP systems to plan for novel temporally extended goals with learning methods. Under this umbrella, there are: methods which learn capabilities that may be difficult to engineer (e.g. a pouring action) [23], those which learn the symbolic representations with which to plan [24], [25], [26], those that integrate perception learning and scene understanding into TAMP [27], [28], and those which attempt to learn search guidance from experience [29], [30], [31].

VI. 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 one 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," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 30, 2020, pp. 440–448.

[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*, vol. 37, no. 10, pp. 1134–1151, 2018. [Online]. Available: https://doi.org/10.1177/0278364918761570

[3] B. Bonet and H. Geffner, "Planning as heuristic search," *Artificial Intelligence*, vol. 129, no. 1, pp. 5–33, 2001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370201001084

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

[5] N. Haghtalab, S. Mackenzie, A. Procaccia, O. Salzman, and S. Srinivasa, "The provable virtue of laziness in motion planning," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 28, 2018, pp. 106–113.

[6] 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.

[7] T. Silver, R. Chitnis, A. Curtis, J. B. Tenenbaum, T. Lozano-Pérez, and L. P. Kaelbling, "Planning with learned object importance in large problem instances using graph neural networks," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 35, no. 13, 2021, pp. 11 962–11 971.

[8] M. Khodeir, B. Agro, and F. Shkurti, "Learning to search in task and motion planning with streams," *IEEE Robotics and Automation Letters*, vol. 8, no. 4, pp. 1983–1990, 2023.

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

[10] S. Brody, U. Alon, and E. Yahav, "How attentive are graph attention networks?" in *International Conference on Learning Representations*, 2022. [Online]. Available: https://openreview.net/forum?id=F72ximsx7C1

[11] C. R. Garrett, R. Chitnis, R. Holladay, B. Kim, T. Silver, L. P. Kaelbling, and T. Lozano-Pérez, "Integrated task and motion planning," *Annual review of control, robotics, and autonomous systems*, vol. 4, pp. 265–293, 2021.

[12] T. Ren, G. Chalvatzaki, and J. Peters, "Extended tree search for robot task and motion planning," 2021. [Online]. Available: https://arxiv.org/abs/2103.05456

[13] C. R. Garrett, C. Paxton, T. Lozano-Pérez, L. P. Kaelbling, and D. Fox, "Online replanning in belief space for partially observable task and motion problems," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020, pp. 5678–5684.

[14] S. Cambon, R. Alami, and F. Gravot, "A hybrid approach to intricate motion, manipulation and task planning," *The International Journal of Robotics Research*, vol. 28, no. 1, pp. 104–126, 2009.

[15] E. Plaku and G. D. Hager, "Sampling-based motion and symbolic action planning with geometric and differential constraints," in *2010 IEEE International Conference on Robotics and Automation*. IEEE, 2010, pp. 5002–5008.

[16] L. P. Kaelbling and T. Lozano-Pérez, "Hierarchical task and motion planning in the now," in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 1470–1477.

[17] M. Toussaint, "Logic-geometric programming: An optimization-based approach to combined task and motion planning," in *Proceedings of the 24th International Conference on Artificial Intelligence*, ser. IJCAI'15. AAAI Press, 2015, p. 1930–1936.

[18] M. Toussaint, K. Allen, K. Smith, and J. Tenenbaum, "Differentiable physics and stable modes for tool-use and manipulation planning," in *Proceedings of Robotics: Science and Systems*, Pittsburgh, Pennsylvania, June 2018.

[19] 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*, vol. 37, no. 10, pp. 1134–1151, 2018.

[20] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel, "Combined task and motion planning through an extensible planner-independent interface layer," in *2014 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2014, pp. 639–646.

[21] F. Lagriffoul and B. Andres, "Combining task and motion planning: A culprit detection problem," *The International Journal of Robotics Research*, vol. 35, no. 8, pp. 890–927, 2016.

[22] K. Hauser and J.-C. Latombe, "Integrating task and prm motion planning: Dealing with many infeasible motion planning queries," in *ICAPS09 Workshop on Bridging the Gap between Task and Motion Planning*. Citeseer, 2009.

[23] Z. Wang, C. R. Garrett, L. P. Kaelbling, and T. Lozano-Pérez, "Learning compositional models of robot skills for task and motion planning," *The International Journal of Robotics Research*, vol. 40, no. 6-7, pp. 866–894, 2021.

[24] T. Silver, R. Chitnis, J. Tenenbaum, L. P. Kaelbling, and T. Lozano-Pérez, "Learning symbolic operators for task and motion planning," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 3182–3189.

[25] J. Loula, K. Allen, T. Silver, and J. Tenenbaum, "Learning constraint-based planning models from demonstrations," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2020, pp. 5410–5416.

[26] M. Diehl, C. Paxton, and K. Ramirez-Amaro, "Automated generation of robotic planning domains from observations," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 6732–6738.

[27] K. Kase, C. Paxton, H. Mazhar, T. Ogata, and D. Fox, "Transferable task execution from pixels through deep planning domain learning," *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 10 459–10 465, 2020.

[28] Y. Zhu, J. Tremblay, S. Birchfield, and Y. Zhu, "Hierarchical planning for long-horizon manipulation with geometric and symbolic scene graphs," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021, pp. 6541–6548.

[29] D. Driess, J.-S. Ha, and M. Toussaint, "Deep Visual Reasoning: Learning to Predict Action Sequences for Task and Motion Planning from an Initial Scene Image," in *Proceedings of Robotics: Science and Systems*, Corvalis, Oregon, USA, July 2020.

[30] D. Driess, O. Oguz, J.-S. Ha, and M. Toussaint, "Deep visual heuristics: Learning feasibility of mixed-integer programs for manipulation planning," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 9563–9569.

[31] L. Xu, T. Ren, G. Chalvatzaki, and J. Peters, "Accelerating integrated task and motion planning with neural feasibility checking," *arXiv preprint arXiv:2203.10568*, 2022.