Robotics: Science and Systems 2023
Daegu, Republic of Korea, July 10-July 14, 2023

Sequence-Based Plan Feasibility Prediction for Efficient Task and Motion Planning

Zhutian Yang[1,*], Caelan Reed Garrett[2], Tomás Lozano-Pérez[1], Leslie Pack Kaelbling[1], Dieter Fox[2]
[1]Massachusetts Institute of Technology, [2]NVIDIA Research

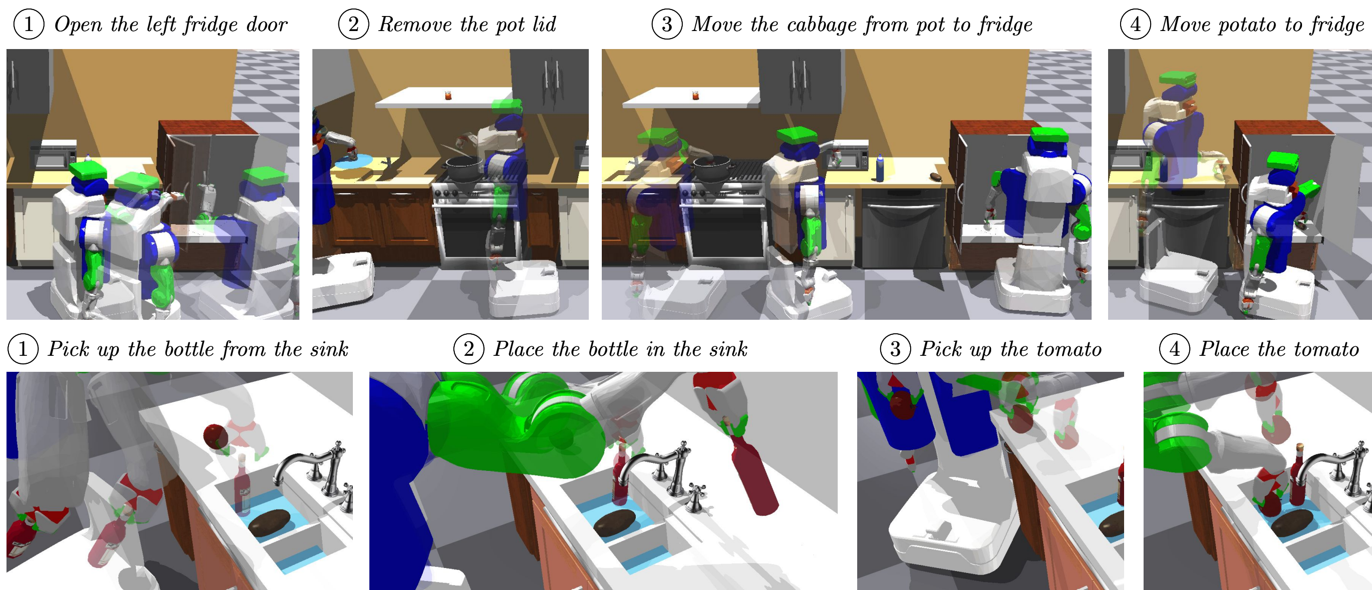


Fig. 1: Example solutions to two task and motion planning problems with complex geometric constraints, for which our PIGINet reduces the planning time by filtering out infeasible task plans considered during planning. **Top:** The goal is for all food to be in the fridge. **Bottom:** The goal is for the tomato to be in the sink. Directly picking and placing the goal objects in both problems is not possible due to obstruction caused by articulated and movable obstacles.

***Abstract*—We present a learning-enabled Task and Motion Planning (TAMP) algorithm for solving mobile manipulation problems in environments with many articulated and movable obstacles. Our idea is to bias the search procedure of a traditional TAMP planner with a learned plan feasibility predictor. The core of our algorithm is PIGINet, a novel Transformer-based learning method that takes in a task plan, the goal, and the initial state, and predicts the probability of finding motion trajectories associated with the task plan. We integrate PIGINet within a TAMP planner that generates a diverse set of high-level task plans, sorts them by their predicted likelihood of feasibility, and refines them in that order. We evaluate the runtime of our TAMP algorithm on seven families of kitchen rearrangement problems, comparing its performance to that of non-learning baselines. Our experiments show that PIGINet substantially improves planning efficiency, cutting down runtime by 80% on problems with small state spaces and 10%-50% on larger ones, after being trained on only 150-600 problems. Finally, it also achieves zero-shot generalization to problems with unseen object categories thanks to its visual encoding of objects. Project page https://piginet.github.io/.**

I. Introduction

Planning for long-horizon robotic behavior in complex environments requires quick reasoning about the impact of the environment's geometry on what high-level plans are feasible. Many task and motion planning (TAMP) [1, 2] algorithms accomplish this by balancing the computational time spent on two processes. One is *task planning*: finding high-level task plans consisting of discrete arguments that achieve the logical conditions specified by the goal. The other is *motion planning*: generating continuous motion trajectories that are collision-free using sampling or optimization.

Balancing between task planning and motion planning is particularly challenging when manipulable obstacles impose additional geometric constraints that makes it hard to find collision-free object placements or arm trajectories. For example, a mobile robot may be tasked with rearranging food items among fridges, cooking pots, and sinks. Doors and other food items may be blocking all paths that reach the goal objects or placement regions, as in the problems shown in Figure 1. In these problems, the number of infeasible candidate task plans increases exponentially as the planning horizon and number of objects both increase. An uninformed TAMP algorithm would waste a substantial amount of time attempting to satisfy many unsatisfiable constraints associated with infeasible task plans, *e.g.* by attempting to solve unsolvable motion planning subproblems, before working on the feasible ones.

Some manipulation approaches deal with obstruction caused

*Work done during an internship at NVIDIA Research

by storage units and containers by assuming that there exist predicates like (`reachable` *sink*) or (`opened` *door*) to help eliminate infeasible task plans during the discrete search process. However, this type of discretization of geometric state is not applicable to real-world situations where regions can be partially occupied, doors can be half open, and multiple doors need to be opened in order to enable reachability. In this paper, we propose an alternative strategy that avoids discretization by generating a diverse set of task plans and pruning out the infeasible task plans using a neural network.

At the core of our framework is ***PIGINet***, a plan feasibility prediction network based on the transformer architecture. Given a candidate task ***P***lan with ***I***mage features of objects, 2) the ***G***oal formula, and 3) the relations and continuous values in the ***I***nitial state, PIGINet outputs a probability that the task plan is feasible. The elements of each action or relation in the initial state–such as text, object poses, and door joint angles– are processed to produce embeddings of the same dimension and fused together to produce each token in the input sequence to transformer encoder. A pre-trained CLIP model [3] is used to generate corresponding text and image embeddings. For each distribution of tasks, the model is trained with up to 4000 plans and plans for up to 600 environments.

We deploy our trained model in a TAMP algorithm that generates a large number of task plans, sorts them by the model's predicted likelihoods of feasibility, and refines them in order of feasibility until a solution is found. We evaluate the success rate and runtime of our learning-enabled TAMP algorithm on unseen problem instances in comparison to a baseline and ablations. Our experiments show that learning to predict task plan feasibility can substantially improve planning performance, cutting down runtime by 80% on problems with a low dimensional state space and 10%-50% on larger ones. It also achieves zero-shot generalization to problems with unseen object categories thanks to its visual encoding of objects.

II. Related Work

Our method builds on prior work in task and motion planning (TAMP), learning to expedite TAMP, and sequence prediction for robot manipulation.

a) Task and Motion Planning: One approach to TAMP first performs a search over high-level task plans and then refines each plan to be refined using sampling or optimization [1]. Two of these algorithms [4, 5] use *diverse planning* techniques [6], which identify multiple distinct plans to produce candidate task plans. Because diverse planners are unaware of the geometry of the world, many candidate plans have the same sources of infeasibility, *e.g.,* due to unreachable objects or regions. As a result, these TAMP algorithms waste time finding continuous values for similar, unsatisfiable motion planning problems associated with those task plans. Many TAMP algorithms contain mechanisms that provide specific feedback [1] to the search over task plan using failed motion queries [7] or unsatisfiable constraint sets [8]. However, these approaches require expensive geometric planning to first identify failed task plans and second generate feedback.

b) Learning to speed up TAMP: TAMP slows down exponentially as the problem horizon and the number of manipulable objects increase. Silver *et al.* [9] developed a Graph Neural Network (GNN) approach that ignores irrelevant objects in table-top rearrangement problems, taking into account object obstruction. Khodeir *et al.* [10] extended this approach to predict which samplers should be prioritized when solving for continuous values. Kim *et al.* [11] learned a cost-to-go heuristic estimator using a relational embedding of the state to guide search. Several learning-for-TAMP approaches learn single-action feasibility classifiers using object poses and relative distances [12], depth image [13, 14, 15], or point cloud [16] encoding of the environment. Compared to work on action feasibility, our approach of classifying feasibility of entire plans enables us to 1) discard infeasible task plans without ever performing motion planning and 2) consider constraints arising from actions late in the plan that restrict choices early in the plan (e.g. whether two doors need to be opened depends on how many objects must be placed inside later and how large they are). Also, we are leveraging, instead of replacing, task planning [10, 17], by providing the learner with sound task plans, easing the learning burden and, in practice, greatly expanding the learner's ability to generalize to varied initial states, goals, and even actions. Furthermore, we are deploying powerful pre-trained language and vision models to represent complex scenes in order to learn models that generalize well from relatively small amounts of training data (150-600 problems for each problem set).

c) Sequence-based modeling for robotic manipulation: We are inspired by recent works that used attention-based neural networks to encode the state, fusing multi-modal inputs, and making object-centric decisions. Zhu *et al.* [18] generate the next action name and action parameters by encoding the states as symbolic and geometric scene graphs and then process them with GNNs. Liu *et al.* [19] predict object poses for semantically meaningful arrangements by encoding text and point-cloud embeddings in the same sequence for a transformer encoder. Blukis *et al.* [20] generate subgoals given language instructions by encoding a sequence of previous subgoals and combining them with a language embedding. Yuan *et al.* [21] learn object embeddings by encoding a sequence consisting of image patches of the whole scene and canonical views of each object in a transformer architecture and then use the learned embedding for querying object spatial relations or a direction for gripper movement. Our work uses similar techniques for merging multi-modal input but deals with more complex spatial relationships between objects, reasoning about obstruction while also dealing with extraneous inputs due to the large state space of our mobile manipulation problems.

III. Problem Formulation

We represent TAMP problems using an extension of the Planning Domain Definition Language (PDDL), a logic-based action language, that supports planning with continuous values [22]. We define a TAMP *domain* $\langle \mathcal{P}, \mathcal{C}, \mathcal{A} \rangle$ by a set of *predicates* $\mathcal{P}$, *constants* $\mathcal{C}$ and *actions* $\mathcal{A}$. *Predicates* and

$\mathcal{I}nit$	$\mathcal{G}oal$
`IsJointTo`(door3, fridge1)	`Closed`(door3)
`IsType`(tomato1, `@food`)	`In`(tomato1, storagespace2)
`Supported`(tomato1, table2, p_0)	`On`(tomato1, table2)

TABLE I: Example initial facts ($\mathcal{I}$) and goal conditions ($\mathcal{G}$), where `@` indicates constants.

Plan skeleton $\hat{\pi}$	*Task plan* π
`pull`(fridge1:door1; a_0, $?a_1$, $?g_1$, $?t_1$)	⟨`pull`, fridge1:door1⟩
`pick`(tomato1; p_0, $?q_2$, $?g_2$, $?t_2$)	⟨`pick`, tomato1⟩

TABLE II: An example plan skeleton and corresponding task $\mathcal{P}$lan used for plan feasibility prediction. Symbols starting with *?* are variables to be assigned during skeleton refinement.

actions can be represented as tuples consisting of a name and a list of typed arguments. The arguments may be (1) discrete, such as object and part names, or (2) continuous, such as object poses, object grasps, robot configurations, object joint angles, and robot trajectories. *Constants* name objects that are useful by all problems in the domain, such as object categories.

A TAMP *problem* $\langle \mathcal{O}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ is defined by a set of *objects* $\mathcal{O}$, a set of *initial literals* $\mathcal{I}$, and a conjunctive set of *goal literals* $\mathcal{G}$. A *literal* is a predicate with an assignment of values to its arguments. The set of initial literals defines a state of the world. Each *grounded action* defines a deterministic transition of the world state. Table I shows some example literals of each construct. Note that "fridge1:door1", "fridge1:space1", and "fridge1" are three different objects in the planning problem since they afford different actions.

A *solution* is a finite sequence of grounded action instances that, when sequentially applied to the initial state $\mathcal{I}$, produces a terminal state where the goal literals $\mathcal{G}$ all hold. During the course of planning, many TAMP algorithms reason about *plan skeletons* $\hat{\pi}$, partial solutions where the continuous arguments are yet to be bound (denoted by the prefix *?*). Some skeletons can be turned into solutions through *refinement* by searching over continuous values for unbound parameters that satisfy the plan's constraints, such as inverse kinematics and collision-free constraints. However, successfully refining a skeleton is not always feasible. A task plan π is a skeleton without continuous arguments, as shown on the right of Table II.

Usually, task plans are produced and refined in order of plan length. Our idea is to refine them in order of plan feasibility, the likelihood that the refinement process can find a set of values for all continuous arguments in $\hat{\pi}$. The role of the plan feasibility predictor f is to take in a task plan π, initial literals $\mathcal{I}$, and goal literals $\mathcal{G}$, and output a score $f(\mathcal{I}, \pi, \mathcal{G})$, as shown in Figure 2. We use f generically as a scoring function for ranking a batch of task plans for refinement. Note that this work addresses traditional TAMP, where the system is assumed to be deterministic and observable. Thus, the planner has full observability of the state, which includes information such as the pose of each object, even if it's occluded from the perspective to the camera or fully enclosed, as well as whether the object is supported by its initial container. Since

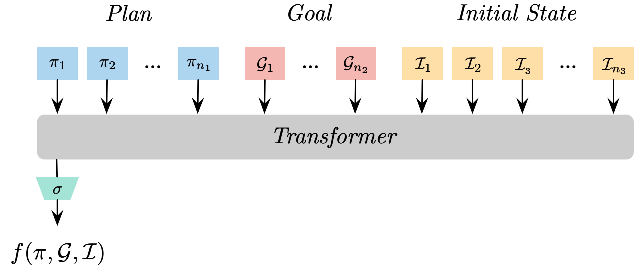


Fig. 2: The input and output of our plan feasibility predictor. **Input** is tokenized high-level actions, goal conditions, and facts in the initial state. **Output** include the predicted likelyhood that the high-level plan can be refined to produce motion trajecories to satisfy all geometric constraints.

the feasibility checker also has access to the full state, we render images from desired camera poses to take advantage of pre-trained computer vision networks that takes in RGB image input, as shown in Figure 3. Here, 6 image viewpoints are used because (a) the kitchen is wide and objects will be too small to be useful for feature extraction when the camera is far away and (b) we need cameras that are looking from top-down views (to see the empty area in a sink or pot) and side views (to see into cabinets and fridges).

IV. LEARNING METHOD

Given a TAMP problem and a task plan, PIGINet first builds a dictionary of embeddings for objects, continuous values, and text using type-specific encoders. Next, it converts each action in the plan, each literal in the goal, and the initial state into tokens and stack them to produce one input sequence for a transformer encoder. The decoded output is the probability of plan feasibility. The architecture is illustrated in Figure 4.

A. Encoding objects, text, and continuous values

Constructing the input sequences requires encoding strategies for elements of different modalities that form the initial state, goal, and candidate plan, including objects, texts, and continuous values.

Objects in the planning problem are represented by images. We use images as an approximation of the geometric state to leverage vision models. Although the planner has full state information including collision geometries, the task planning process normally would not consider geometry as it reasons over symbolic states. We 1) render RGB images of the scene from C cameras and query instance segmentation of each object $\{\{x_{c,o}\}_{c=1}^{C}\}_{o=1}^{|O|}$, 2) extract their features with a pre-trained vision network f_{img}, 3) concatenate the features from different cameras for each object, and 4) reduce the dimension of the resulting feature vectors with a three-layer Multi-Layer Perceptron (MLP),

$$g_{img}(o) = MLP_{img}([f_{img}(x_{1,o}); \ldots; f_{img}(x_{C,o})]).$$

Text used in the domain, which includes the names of predicates and actions as well as constants, is represented

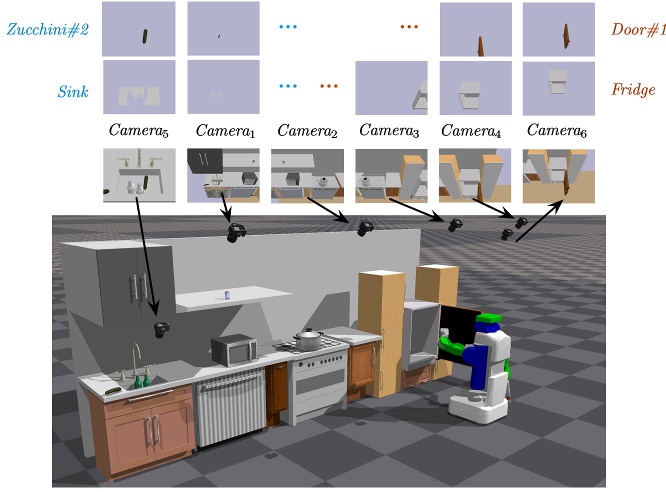


Fig. 3: Example camera poses and segmented images rendered in PyBullet. The images with only background color are omitted. *Cameras* 1-4 form an array that covers the whole kitchen, while *Cameras* 5-6 look at goal-related regions.

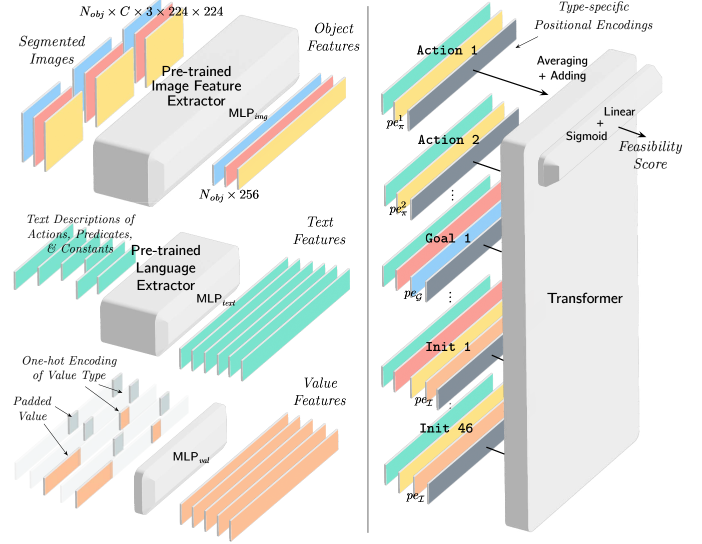


Fig. 4: PIGINet architecture. **Left**: Encoding of objects, values, and text into feature vectors of the same length. **Right**: Construction of input sequence for transformer encoder from lists of variable lengths from the plan, goal, and initial state.

as feature vectors with the same dimension as object and value embeddings. There is a fixed number of such words in a domain, such as `SupportedBy`, `Pull`, and `@bottle`. For a word w, we first describe it using a colloquial English phrase or sentence s. Rephrasing helps the network deal with out-of-distribution names in the domain description file such as `isjointto`, which is rewritten to "*this is a joint of that object*". Then, we encode it with a pre-trained language embedding network f_{text} and transform the output feature through a linear and a ReLU layer,

$$g_{text}(w) = MLP_{text}(f_{text}(s)).$$

Continuous values in the initial literals, such as object poses $p = (x, y, z, yaw)$ and door joint angles $a = (\theta,)$, are each treated as typed tuples. We define a fixed set of value types $\mathcal{T}$ and their corresponding tuple length $\mathcal{L}$. First, we normalize the values to fall in the range $[-1, 1]$. For example, the x, y, z values are normalized with respect to the bounding box of the world. The door angles, originally are in $[0, \mathcal{L}_{upper}]$ where 0 means the door is fully closed and $L_{upper} \leq \pi$, are normalized to $[0, 1]$. Then, we zero-pad tuples of varying lengths to become feature vectors of the same dimension $\sum_{i=0}^{|\mathcal{T}|} L_i$. For example, if value $v = (v_1, ..., v_{L_j})$ has type $\mathcal{T}_j$, the resulting vector $\tilde{v}$ has zero in all positions except from $\sum_{i=0}^{j-1} \mathcal{L}_i$ to $\sum_{i=0}^{j} \mathcal{L}_i$. Then, we concatenate a one-hot encoding of $\mathcal{T}_j$ with $\tilde{v}$. For example, if $p_0 = (x, y, z, yaw)$ and $a_0 = (\theta,)$ are the only values used in the initial literals, $\mathcal{T} = [1, 2]$, $\mathcal{L} = [4, 1]$, the processed features will be

$$p_0 : [\underbrace{1, 0}_{\tau=1,\text{ type is pose}}, \underbrace{\hat{x}, \hat{y}, \hat{z}, \hat{yaw}}_{\textit{normalized value}}, 0],$$

$$\text{and } a_0 : [\underbrace{0, 1}_{\tau=2,\text{ type is angle}}, 0, 0, 0, 0, \underbrace{\hat{\theta}}_{\textit{normalized value}}].$$

Lastly, we process the resulting feature vectors with a linear and a ReLU layer,

$$g_{val}(v) = MLP_{val}([\textit{one-hot}(j, |\mathcal{T}|); \tilde{v}]).$$

B. Transformer encoder for Plan, Images, Goal, Init

We view each literal in $\mathcal{I}nit$ and $\mathcal{G}oal$, as well as each action in π, as one list of multi-modal elements $\mathbf{z} = [z_1, ..., z_m]$ with element types $\mathbf{t} = [t_1, ..., t_m]$. For literals, $m \in [2, 4]$; for actions, $m \in [1, 2]$. We produce a token out of the list by mapping each element of the tuple to its feature embedding, averaging the embeddings, and then adding a *positional encoding*. We choose to average the embeddings, as opposed to concatenating them, in order to handle tuples of varying length. In practice, the two aggregation methods yield similar training performance. *Positional encodings* are used by transformer architectures as an addition to each input tokens because the order matters for sequence-based models. Each action in π has a different sinusoidal positional encoding $pe_\pi^{1:k}$ where k is the number of actions. For the set of literals in $\mathcal{G}$ and in $\mathcal{I}$, the sequence of constituting literals doesn't matter, so the positional encoding could denote their token type. All literals in $\mathcal{G}$ have the same learned positional encoding $pe_\mathcal{G}$, and all literals in $\mathcal{I}$ have the same learned positional encoding $pe_\mathcal{I}$. In summary, the process of translating a list $\mathbf{z}$ into an input token x_k is as follows, where $g_{t_i} \in [g_{img}, g_{text}, g_{val}]$ are defined previously and $type(\mathbf{z}) \in [\pi, \mathcal{G}, \mathcal{I}]$,

$$x_k = h(\mathbf{z}, \mathbf{t}) = \frac{1}{m} \sum_{i=1}^{m} g_{t_i}(z_i) + pe^k_{type(\mathbf{z})}.$$

The transformer encoder takes in the sequence of tokens x representing π, $\mathcal{G}$, and $\mathcal{I}$,

$$x_{1:n} = [h(\pi_1), ..., h(\pi_{n_1}), h(\mathcal{G}_1), ..., h(\mathcal{G}_{n_2}), h(\mathcal{I}_1), ..., h(\mathcal{I}_{n_3})].$$

Embeddings of initial literals are dropped uniformly at random when the length of the sequence exceeds the max sequence length n. We use multi-headed attention to enable each position in the sequence to attend to others, except for within the plan tokens. The plan tokens use a causal mask, to build in the bias that the feasibility of one action doesn't depend on future actions. In practice, models trained with causal-plan mask and full mask achieve similar learning performance. Using three layers of residual attention blocks, we get an output y with the same length as input x,

$$y_{1:n} = \mathit{Transformer}(x_{1:n}).$$

We keep only the first position of the output and add linear and sigmoid layers to produce a nonnegative feasibility score,

$$f(\pi, \mathcal{G}, \mathcal{I}) = \mathit{MLP}_{out}(y_1).$$

C. Training

We train PIGINet using the binary cross-entropy classification loss between the prediction and the label. We add a positive weight to the loss function according to the ratio of negative to positive training examples. The models are trained until prediction accuracy converges on the validation set. We use the confidence of the prediction as feasibility score. The whole architecture is trained end-to-end.

V. PLANNING ALGORITHM

We developed a new PDDLStream [22] algorithm, *batch-sorted*, that uses our PIGINet for feasibility prediction by feeding it a large number of candidate task plans for scoring. Algorithm 1 gives the pseudocode for the main algorithm BATCH-SORTED-TAMP. Similar to the *focused* algorithm [22], it lazily instantiates a set of free action parameters X, that stand in for actual continuous values, using the available sampling operations. Namely, it recursively creates free parameters that optimistically represent the possible output of the sampling operations. The subroutine NEW-PARAMETERS increases the depth of the recursive parameter instantiation.

BATCH-SORTED-TAMP repeatedly searches for k distinct plan skeletons to make up a single skeleton batch Π_k, where $k \geq 1$. New plans are identified by performing a forward plan-space search FORBID-SEARCH that forbids any previously identified plans Π from the search's open list [23]. After k attempts, the batched plans Π_k are scored using the learned feasibility predictor $f(\pi, \mathcal{G}, \mathcal{I})$. Plans that are predicted to have feasibility < 0.5 are discarded. The rest are sorted in decreasing order of predicted likelihood of feasibility. The algorithm attempts to refine each plan in order using sampling via REFINE-PLAN. This involves searching over combinations of sampler output values that bind each free parameter and jointly satisfy the preconditions of each step in the plan. The first fully-bound plan π_* is returned as a solution.

When running without a feasibility predictor (*i.e.* $f(\pi, \mathcal{G}, \mathcal{I}) = 1$), BATCH-SORTED-TAMP is a *complete* algorithm for PDDLStream planning, assuming that FORBID-SEARCH is a complete discrete search. This follows from the fact that the set of parameters X defines a finite discrete search subproblem. FORBID-SEARCH will eventually enumerate all solutions to this problem. Once exhausted, NEW-PARAMETERS expands the subproblem, admitting more solutions. We attempt to refine each solution using REFINE-PLAN, which can be done indefinitely, for example, in a parallel thread. When a feasibility predictor is present and also produces false negatives, we can obtain a complete algorithm by, rather than rejecting them, refining them at a lower computation rate. Finally, because these algorithms are complete for PDDLStream, they are also *probabilistically complete* with respect to the underlying manipulation problem if the samplers satisfy some sample coverage properties [24].

Algorithm 1 Batch Sorted TAMP Plan Prediction

```
Require: Feasibility predictor: f(π, G, I) → [0, 1]
 1: procedure BATCH-SORTED-TAMP(O, I, G, A; k)
 2:     X ← ∅                          ▷ Initialize plan free parameters
 3:     Π ← ∅                          ▷ Initialize identified plans
 4:     while True do
 5:         Π_k ← ∅                    ▷ Initialize batch of at most k plans
 6:         for i ∈ {1, ..., k} do
 7:             π ← FORBID-SEARCH(O, X, I, G, A; Π)
 8:             if π ≠ None then       ▷ Identified a new plan
 9:                 Π ∪= {π}
10:                 Π_k ∪= {π}
11:             else                   ▷ Infeasible: add more parameters
12:                 X ∪= NEW-PARAMETERS(O, I, X)
13:         P = {⟨f(π, G, I), π⟩ | π ∈ Π_k}
14:         for ⟨p, π⟩ ∈ reversed(sorted(P)) do
15:             if p ≥ 0.5 then        ▷ Filter plans
16:                 π_* = REFINE-PLAN(π; I, G)
17:                 if π_* ≠ None then
18:                     return π_*     ▷ Return the bound solution
```

VI. DATA GENERATION

We study a collection of tasks that require moving objects to target surfaces or storage units, using actions `move` (robot motion), `pick` and `place` (prehensile object manipulation), and `pull` (operating single degree-of-freedom mechanisms such as doors and drawers). We use *problem set* to refer to a set of problems that are similar in object types, initial object relations, and goal formulas, but vary in scene layout, object assets, and initial world configuration.

We first experiment with a simple setting, the *Fridge (FG)* problems, where the scene contains a small set of objects so a single camera can capture all objects. Then we experiment in a larger setting, the *Kitchen (KC)* problems, where the problems contain extraneous articulated and movable objects. Because the kitchens are wide, there are 6 simulated cameras: 4 arranged in a line pointing to the front face of the long kitchen space and 2 close-up cameras pointing at the sink and cooking pot from a top-down view. Together, they capture the shape and visibility of objects. In practice, we observe that

models trained without images from the close-up cameras tend to perform not as well.

We briefly describe the problem sets as follows. For detailed descriptions of all seven problems and differences in PIGINet hyper-parameters for training on them, please see Appendix.

A. The Fridges problem sets

There are fridges on top of tables and food items that need to be in the fridge. Each fridge door is either closed 50% of the time, or open at a position sampled uniformly at random across its joint limits. We also randomly sample the pose of objects, the height of tables, and the initial base configuration of a PR2 robot, as shown in Figure 5(a). We used articulated URDFs from the PartNet-Mobility dataset [25] and food meshes sourced online. We generated the scenes using seven fridge assets, nine food assets, and 11 table assets. Overall, the problems contain 1-2 movable objects, 1-2 surfaces, 1-2 storage units, and 1-5 doors. Successful task plans to the problems include 1 pick-and-place and 0-2 pulls.

B. The Kitchens problem sets

The *Kitchen* problems have similar randomized properties as the *Fridges* problem set: the poses of movable objects and positions of doors for storage units are randomly sampled. The kitchen environments have different compositions of furniture and appliances but the same number of movable objects, including two food items, two bottles, and two pill bottles in each scene, as shown in Figure 5(b). The objects are drawn from an asset library consisting of seven assets for each object category, except for the small upper cabinets, of which we used three. The manipulable objects include all goal-related movables and doors, with an additional movable object and one storage unit (with unrelated doors) when the goal region is a storage unit. By adding irrelevant objects to the problem, we test PIGINet's ability to choose plans that don't involve those objects based on the goal and initial state. The other objects in the scene are used as static collision bodies. Overall, the problems contain 2-5 movable objects, 2-4 surfaces, 0-2 storage units, and 0-5 doors. Successful task plans to the problems include 1-3 pick-and-place and 0-2 pulls.

C. Data collection

After sampling a scene and a goal, we run the *batch-sorted* planning algorithm without any feasibility checking to generate up to 100 task plans for the problem. The planner refines each plan until it finds one solution as a positive example. All attempted but failed task plans are labeled as negative examples, as are task plans involving task-irrelevant objects. Labels are noisy, as they are estimates of plan feasibility. Deciding plan feasibility is NP-hard (via a 3-SAT reduction), making obtaining exact labels computationally intractable. Thus, we assign a label to be positive if the planner found a solution within a timeout. We render the segmented images in the PyBullet simulator offline, which includes segmented object links such as doors. If an object is occluded, its associated images will consist entirely of the background color. We save the problem $\langle \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$ along with corresponding images and one task plan as one data point.

We generated a dataset of 600 problems for problem set `table-to-fridge` and `fridge-to-fridge`; 500 for `counter-to-storage` and `counter-to-pot`, 250 for `pot-to-storage`, and 150 for `counter-to-sink` and `sink-to-storage` each. The names of the problem set describe the initial placement region(s) of goal objects and their destination placement region(s). We both train our models and evaluate the integrated planners on individual problem sets, except for the last one where problems involving two different kinds of goals, *i.e.,* moving to the sink and from the sink, are trained together and tested separately. For training each model, we divide the data with 9-to-1 training/validation split to evaluate training performance and an additional 50 test problems for each task to evaluate the improvement of planning performance. During training, we augment the images with random crops, rotations, shifts, warps, color jittering, blurring, and grayscale transformations. We used a pre-trained CLIP model [3] for image and text embedding. The dataset of problems and solutions will be released upon acceptance of the paper, along with the code for generating kitchen layout.

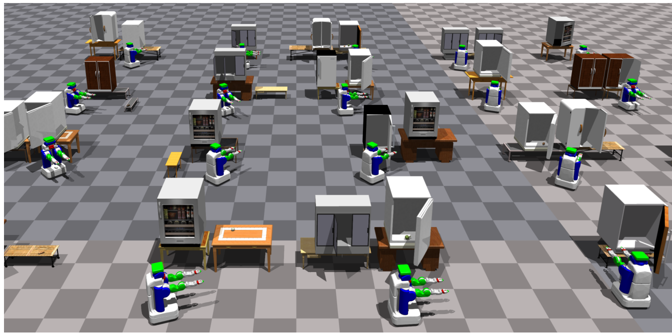
VII. EXPERIMENTS

We carried out experiments to answer the following three questions: (1) **Efficiency**: Can PIGINet improve planning speed without sacrificing planner success rate? (2) **Generalization**: Can a trained model make accurate predictions in problems with unseen objects? (3) **Ablation**: If we take away parts of the input, can PIGINet still make accurate predictions?

A. Planner performance

First, we investigate the effectiveness of PIGINet for speeding up planning. We compared the planning time of three different planner ablations of Algorithm 1. 1) `Baseline` is a learning-free planner that always returns $f(\pi, \mathcal{G}, \mathcal{I}) = 1$, attempting to refine every skeleton in the order of ascending plan length. 2) `PIGI` sorts the plans with the probability generated by our PIGINet. 3) `Oracle` refines only those task plans that have been logged to be feasible offline, serving as an upper bound on possible performance.

We test each planner on 30-50 unseen problems for each problem set and record the runtime breakdown as well as the number of infeasible task plans the planner spent on refining before producing a solution. We set a 3-60 second timeout for producing diverse task plans using a modified FastDownward planner, depending on the average time it takes to generate a feasible task plan among the batch. We set a 30-60 second timeout on refining each task plan depending on the difficulty of the problem set. We set no total timeout so that all problems are solved by all planners eventually. We set no timeout on batch-producing feasibility scores as they take relatively little time, limited only by the GPU memory available for loading up a trained PIGINet and the number of candidate task plans. As the number of objects and problem horizon increases among problem sets, the number of plans grows exponentially.



(a) Example scenes in the *Fridges* problem set.

(b) Example kitchen layouts in the Kitchens problem set.

Fig. 5: Example procedurally generated environment for complex rearrangement problems. To create TAMP problems used for training and testing, we 1) sample a scene, 2) sample a goal, 3) create the initial literals, 4) run a TAMP planner to find a solution, 5) label the task plan associated with the solution as a feasible plan and record the previous task plans that failed due to timeout during refinement as infeasible plans, 6) render segmented RGB images in simulation.

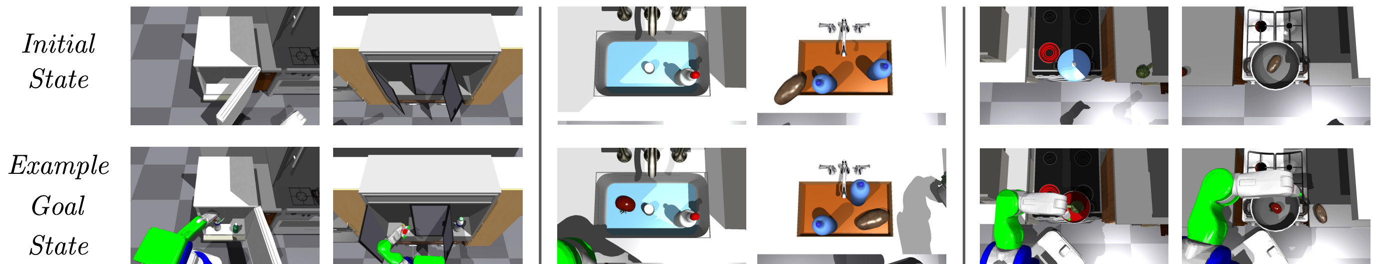


Fig. 6: Example sources of obstruction in our problem sets. **Fridges** may have one to three doors that are partially or fully closed. **Sinks** may be cluttered so object(s) need to be moved away to pick up object from it or place into it. **Cooking pots** may be occluded by a lid or an object inside, which need to be moved away if there aren't enough space to fit the goal object.

So does the size of PIGINet, since there are more images from different camera poses and more initial literals to encode.

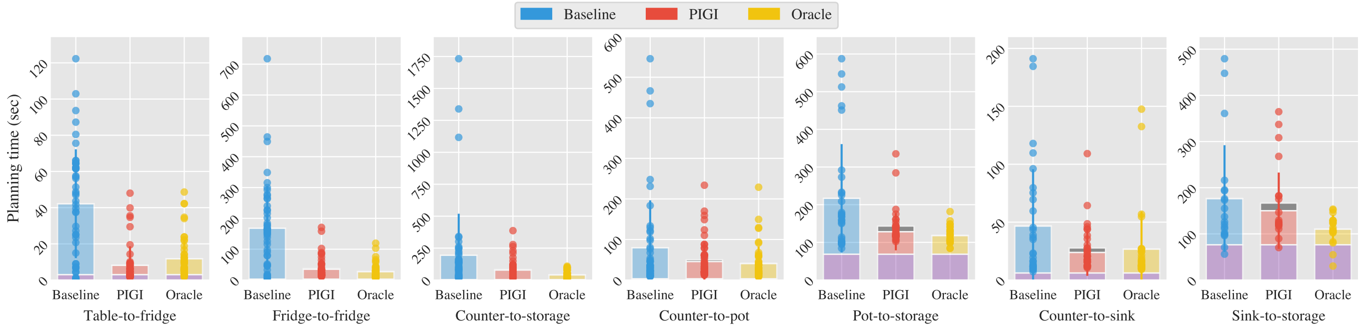
Figure 7(a) shows that PIGINet is able to cut down planning time across all seven problem sets. It reduced runtime by 80% on the *Fridges* dataset and refinement time by 50% on $Kitchens$ problems. Given 50 or 100 task plans, PIGINet usually cuts down the number of infeasible task plans to 1 - 4 and finds a solution after sampling one or two false positive plans, as shown in Figure 7(b). From PIGINet's predictions on individual test problems, we observe that it helps with planning the most when there are multiple doors and two of them need to be opened in order to place two objects inside. In those cases, it cuts down dozens of infeasible task plans and ones that the manipulate irrelevant doors. Note that the improvement in the number of false positive plans is larger than that in planning time. This disparity is because the shorter plans prioritized by the `Baseline` planner impose fewer constraints and thus need less time to be proven infeasible during refinement than those longer ones sometimes ranked highly by PIGINet. We think PIGINet's less decisively improved performance on the *Sink-to-storage* is the result of only being able to collect a limited number of training examples (150 problems for each), relative to the large network size required to handle up to 9 movable and articulated objects.

B. Generalizing to novel objects

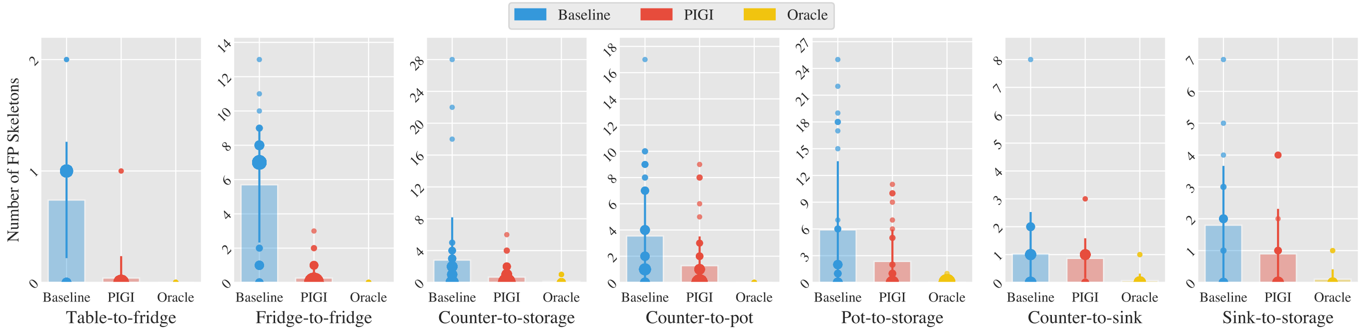
We show that PIGINet can generalize to problems with unseen objects. For a chosen asset, we held out a test set consisting of problems where the object appeared in the goal condition and train a model with the remaining problems. We compare the model's prediction accuracy on the validation set (with seen assets) and the held-out test set (with unseen assets). We experimented with leaving out three food assets and three fridge assets individually. Figure 8(a) shows that the models have equally good accuracy on the unseen object assets as the seen ones (scattered dots align around the diagonal line). We also created a test set where the goal is to move staplers from fridge to fridge. We used five stapler assets from PartNet-Mobility dataset. We see similar improvements in planning time, confirming that PIGINet models can generalize to problems with different object categories and shapes. We attribute this impressive generalization ability to 1) the use of a large pre-trained network CLIP and 2) data augmentation techniques used during training, which makes the model less sensitive to colors and texture.

C. Ablation studies

Finally, we studied how the PIGINet's prediction accuracy is affected when we remove different components of the multi-modal encoding. We compare the models' prediction

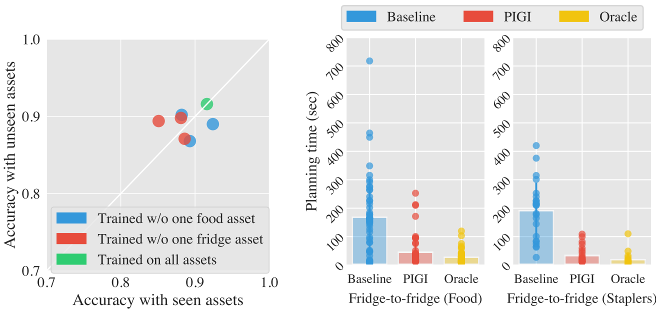


(a) The runtime breakdown of PIGINet-enabled planners compared to a non-learning `Baseline` and a clairvoyant `Oracle`. The purple bars indicate average task planning time by FastDownward for generating hundreds of candidate task plans in those problem sets. The gray bars indicate inference time running the candidate task plans through the PIGINet.



(b) The number of false positive (FP) task plans refined. The size of scattered dots are proportional to density.

Fig. 7: Evaluation of the PIGINet's ability to reduce planning time, after being trained on individual problem sets and evaluated on 30-50 unseen test problems. Scatter plot points are data points. Bar heights are the mean values with whiskers indicating standard deviation. Each subplot's y-axis is scaled differently. Results show that using PIGINet for feasibility-based plan sorting a) significantly reduces planning time and b) enables the planner to find a solution mostly within three refinement iterations.



(a) Leave-one-out accuracy. (b) Test runtime for the stapler task.

Fig. 8: Evaluation of PIGINet's ability to generalization to unseen objects. **Left:** Each dot indicates one held-out experiment, with x and y values showing the model's prediction accuracy on problems with seen or unseen assets. **Right:** The planning time reduction by PIGINet is similar when goal objects changed from food to novel category staplers.

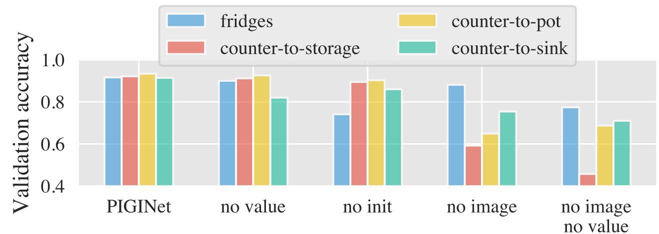


Fig. 9: Classification accuracy on the validation set across ablations of our method when different input components are removed. Overall, PIGINet taking full input performs the best.

accuracy on the validation set for four problem sets after the loss converged. The `fridges` model was trained with both families of problems in the *Fridges* problem sets. Figure 9 shows that PIGINet with all input modalities achieves the best prediction accuracy. Models without continuous values perform almost as well. As for images and initial literals, it seems that discarding initial literals doesn't affect as much when there are a lot of image inputs (due to the 5-6 camera viewpoints used in *Kitchens* problems), compared to the *Fridges* problems where only one image viewpoint is used and objects are scaled differently in the images in order to be sufficiently large. That image cropping process causes PIGINet to lose information about the geometric relationships between objects, which initial literals provide, with literals like `IsJointTo`(*door1*, *fridge*) and `SupportedBy`(*Bottle1*, p_0, *Sink*). These experiments confirmed our hypothesis that

images and initial literals contain redundant information as representation of geometric state. This is also good news for using PIGINet for real-robot settings where privileged initial literals that assume full observability can be hard to obtain.

VIII. CONCLUSION

We developed a novel learning-enabled TAMP algorithm that consists of 1) a multi-modal transformer for predicting feasibility of a batch of candidate task plans given the initial state and goal and 2) a TAMP planner that refines the task plans in the order of predicted feasibility. Our method reduces planning time on complex rearrangement problems with articulated and movable obstacles, where uninformed planners suffer from wasted refinement efforts. PIGINet also achieves zero-shot generalization across unseen movable object categories thanks to its visual encoding of objects.

ACKNOWLEDGEMENTS

We would like to thank Weiyu Liu, Wentao Yuan, Valts Blukis, Danfei Xu at NVIDIA Research, and Jiayuan Mao from Learning and Intelligent Systems Group at MIT for helpful discussions on the project. We gratefully acknowledge support from AI Singapore AISG2-RP-2020-016; NSF grant 2214177; from AFOSR grant FA9550-22-1-0249 and from ARO grant W911NF-23-1-0034.

REFERENCES

[1] Caelan Reed Garrett, Rohan Chitnis, Rachel Holladay, Beomjoon Kim, Tom Silver, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Integrated Task and Motion Planning. *Annual Review of Control, Robotics, and Autonomous Systems*, 4, 2021.

[2] Joaquim Ortiz-Haro, Erez Karpas, Michael Katz, and Marc Toussaint. A conflict-driven interface between symbolic planning and nonlinear constraint solving. *IEEE RA-L*, 7(4), 2022.

[3] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *ICML*. PMLR, 2021.

[4] Tianyu Ren, Georgia Chalvatzaki, and Jan Peters. Extended tree search for robot task and motion planning. *arXiv preprint arXiv:2103.05456*, 2021.

[5] Joaquim Ortiz-Haro, Erez Karpas, Marc Toussaint, and Michael Katz. Conflict-directed diverse planning for logic-geometric programming. *ICAPS*, 2022.

[6] Michael Katz, Shirin Sohrabi, and Octavian Udrea. Top-quality planning: Finding practically useful sets of best plans. In *AAAI*, volume 34, 2020.

[7] Neil T Dantam, Zachary K Kingston, Swarat Chaudhuri, and Lydia E Kavraki. An incremental constraint-based framework for task and motion planning. *IJRR*, 37(10), 2018.

[8] Caelan Reed Garrett. *Sampling-Based Robot Task and Motion Planning in the Real World*. PhD thesis, Massachusetts Institute of Technology, 2021.

[9] Tom Silver, Rohan Chitnis, Aidan Curtis, Joshua B Tenenbaum, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Planning with learned object importance in large problem instances using graph neural networks. In *AAAI*, volume 35, 2021.

[10] Mohamed Khodeir, Ben Agro, and Florian Shkurti. Learning to search in task and motion planning with streams. *arXiv preprint arXiv:2111.13144*, 2021.

[11] Beomjoon Kim and Luke Shimanuki. Learning value functions with relational state representations for guiding task-and-motion planning. In *CoRL*. PMLR, 2020.

[12] Andrew M Wells, Neil T Dantam, Anshumali Shrivastava, and Lydia E Kavraki. Learning feasibility for task and motion planning in tabletop environments. *IEEE RA-L*, 4(2), 2019.

[13] Danny Driess, Ozgur Oguz, Jung-Su Ha, and Marc Toussaint. Deep visual heuristics: Learning feasibility of mixed-integer programs for manipulation planning. In *2020 ICRA*. IEEE, 2020.

[14] Lei Xu, Tianyu Ren, Georgia Chalvatzaki, and Jan Peters. Accelerating integrated task and motion planning with neural feasibility checking. *arXiv preprint arXiv:2203.10568*, 2022.

[15] Smail Ait Bouhsain, Rachid Alami, and Thierry Simeon.

Learning to predict action feasibility for task and motion planning in 3d environments. *hal-03808885*, 2022.

[16] Suhan Park, Hyoung Cheol Kim, Jiyeong Baek, and Jaeheung Park. Scalable learned geometric feasibility for cooperative grasp and motion planning. *IEEE RA-L*, 2022.

[17] Danny Driess, Jung-Su Ha, and Marc Toussaint. Learning to solve sequential physical reasoning problems from a scene image. *The International Journal of Robotics Research*, 40(12-14):1435–1466, 2021.

[18] Yifeng Zhu, Jonathan Tremblay, Stan Birchfield, and Yuke Zhu. Hierarchical planning for long-horizon manipulation with geometric and symbolic scene graphs. In *ICRA*. IEEE, 2021.

[19] Weiyu Liu, Chris Paxton, Tucker Hermans, and Dieter Fox. Structformer: Learning spatial structure for language-guided semantic rearrangement of novel objects. In *2022 ICRA*. IEEE, 2022.

[20] Valts Blukis, Chris Paxton, Dieter Fox, Animesh Garg, and Yoav Artzi. A persistent spatial semantic representation for high-level natural language instruction execution. In *CoRL*. PMLR, 2022.

[21] Wentao Yuan, Chris Paxton, Karthik Desingh, and Dieter Fox. Sornet: Spatial object-centric representations for sequential manipulation. In *CoRL*. PMLR, 2022.

[22] Caelan R. Garrett, Tomás Lozano-Pérez, and Leslie P. Kaelbling. PDDLStream: Integrating Symbolic Planners and Blackbox Samplers. In *ICAPS*, 2020.

[23] Caelan Garrett, Michael Katz, Tomás Lozano-Pérez, Leslie Kaelbling, and Shirin Sohrabi. Diverse planning to cover planning-time uncertainty about initial state. 2020.

[24] Caelan Reed Garrett, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Sampling-based methods for factored task and motion planning. volume 13, 2017. ISBN 9780992374730. doi: 10.1177/0278364918802962. URL https://arxiv.org/abs/1801.00680.

[25] Fanbo Xiang, Yuzhe Qin, Kaichun Mo, Yikuan Xia, Hao Zhu, Fangchen Liu, Minghua Liu, Hanxiao Jiang, Yifu Yuan, He Wang, et al. Sapien: A simulated part-based interactive environment. In *CVPR*, 2020.

Appendix

A. Planning problem sets

The *Kitchens* problem sets are as follows:

(i) `table-to-fridge`: the food is initially on the table. Successful plans involve 1 pick-and-place and 0-2 pulls.

(ii) `fridge-to-fridge`: the food is in another fridge. Successful plans involve 1 pick-and-place and 0-4 pulls.

The *Kitchens* problem sets are as follows.

(iii) `counter-to-storage`: two instances of food or bottles are to be stored in the fridge or an upper cabinet. Successful plans involve 2 pick-and-place and 0-2 pulls.

(iv) `counter-to-pot`: one food item is to be in the pot. The placement path may be blocked by a lid on the pot and/or an object that's placed inside the pot. The object inside doesn't have to be removed if there is still enough room to fit the target object in. Successful plans involve 1-3 pick-and-place and 0-2 pulls.

(v) `pot-to-storage`: two food items are to be in the fridge and one of them is initially inside the pot, which may be covered by the lid. Successful plans involve 1-3 pick-and-place and 0-2 pulls.

(vi) `sink`: this is a union of two problem sets involving two different types of goals: (a) `counter-to-sink`: one food item is to be in the sink, which is occupied by one or two obstacles inside; successful plans involve 1-3 pick-and-place. (b) `sink-to-storage`: two food items are to be in a storage unit, while at least one of the food item is originally inside the sink with one or two other obstacles; successful plans involve 2-3 pick-and-place and 0-2 pulls. For both sets, the manipulable objects include the target objects, obstacles in the sink, and one extra movable object on the counters. There are 150 problems from each set for training. For evaluating planning time, 30 problems from `counter-to-sink` and 20 problems from `sink-to-storage` are used.

B. Model hyper-parameters

We used images from five camera poses for problem set (iii, iv) and six camera poses for (v-vi) To efficiently train the transformers, we used a max sequence length of 56 for problem sets (v, vi) and 32 for the others.