Learning Feasibility and Cost to Guide TAMP

Christopher Bradley[1] and Nicholas Roy[1]

Massachusetts Institute of Technology, Cambridge, MA 02139, USA,
{cbrad, nickroy}@csail.mit.edu

Abstract. Recent work in Task and Motion Planning (TAMP) has enabled a new class of algorithms that can better take advantage of off-the-shelf samplers and solvers to find solutions to sub-problems in a task plan, such as motion between configurations, or inverse kinematics solutions. However, not all sub-problems are equally valuable. Existing planners typically rely on heuristics to determine which sub-problem to attempt to solve next, unable to reason about the expected cost of doing so in the broader context of the full plan. In this work, we present a novel approach for TAMP, utilizing learned models to inform when to attempt to solve potentially expensive sub-problems. We test our approach in two simulated domains, as well as on a real Panda robot, showing improvement in planning and execution time compared to a heuristic driven baseline.

1 INTRODUCTION

We aim to enable an autonomous agent to efficiently find low-cost plans for high-dimensional, long-horizon TAMP problems using learned models to guide planning. Consider, for example, a mobile robot attempting to 'cook' multiple objects in a kitchen environment. Solving this problem involves considering both the discrete sequence of actions (e.g., 'pick up object 'A', navigate within the room, then place 'A' on a new platform') and the constrained continuous and discrete parameters of those actions (a reachable grasp on 'A', a collision-free trajectory to move between configurations, a platform to place on, etc). A common approach in TAMP involves defining samplers or solvers for relevant sub-problems (e.g., a grasp sampler or collision-free motion planner for trajectories between robot configurations), then combining sub-problem solutions into a complete plan [1].

One major challenge in this strategy is determining when to attempt to solve a particular sub-problem that is a part of one action plan, versus spending computation time solving a different sub-problem from a separate, potentially more feasible, action plan. Given the stochasticity inherent in certain sub-problems (as in a grasp sampler), and because the outcome of sub-problems may depend on their inputs, we must also consider how the solution to one impacts another. Particularly in settings where we fail to solve certain sub-problems frequently (e.g., attempting to place an object on a cluttered platform), repeated failed attempts can make planning intractable. Unfortunately, knowing ahead of time which sub-problems have feasible (or optimal) solutions is as hard as the original planning problem itself. There is no practical way to avoid periodically attempting to solve sub-problems that do not actually have feasible solutions, but we would like to be able to identify ahead of time the expected cost of doing so, and subsequently minimize wasted computational effort.

In this work, we propose two contributions. The first is a method to compute the expected cost of attempting to solve the relevant sub-problems within a high-level plan in a TAMP problem. Specifically, for each sub-problem (like grasp sampling or IK), we train a model to predict outcomes (success or failure) and costs (in both planning and execution time) for different inputs. We propose a method by which these models can be evaluated to return the expected cost of a full high-level plan, without having to immediately solve each sub-problem. Second, we propose a novel planning approach which uses stochastic search techniques to account for potential inaccuracies in our models, as well as the need to potentially re-solve upstream sub-problems. We demonstrate an improvement in planning time on two different, simulated platforms over a heuristic driven baseline, as well as on a real robot [2].

2 BACKGROUND

The combined TAMP problem jointly considers elements of high-level task planning [3, 4] and low-level motion planning [5] in an attempt to solve hybrid discrete/continuous, multi-modal planning problems [6]. Solutions for TAMP problems take the form of a sequence of parameterized actions $\pi = [a_1, a_2, ..., a_n]$ that define a plan, where parameters satisfy each action's constraints [6]. One approach for representing a TAMP problem —which we use in this work— is an extension of the Planning Domain Definition Language (PDDL) called PDDLStream [1].

A PDDLStream problem (P, A, S, O, I, G) is specified as sets of predicates P, actions A, streams S, objects O, an initial state I, and a goal state set G. The initial and goal states of a PDDLStream problem are sets of facts: instances of boolean functions called predicates $p(x) \in P$, which are parameterized by tuples of objects $x \in O$. For example, the fact that certifies if the robot is at a given pose is an instantiation of the predicate $AtPose\ ?p$, and is either true of false for different pose objects $?p$. Actions $a \in A$ are defined by two sets of predicates: pre and post conditions, and are parameterized by object tuples x. For a given input x, if the pre-conditions evaluate to true, the action is legal, and the post-conditions specify which facts will change value in the subsequent state.

For certain actions, preconditions may include facts that are either cumbersome or impossible to add to the domain. For example, it is unclear how one would enumerate all possible configuration objects for a 20-DOF robot without creating a potentially intractably large problem. To account for this, PDDLStream problems include object generators called *streams* $s \in S$, which allow the planner to represent sub-problems relevant to the problem. Streams consist of: sets of (1) input and (2) output objects, (3) domain predicates which must be true in the input, (4) action predicates to be certified if the queried stream is successful, and (5) an external function that is called when the stream is queried. When an action has a precondition which can only be certified by a stream, that stream can be queried in an attempt to solve the associated sub-problem, and determine if said precondition can be certified. Note that a stream may need to be queried many times before generating an output that jointly satisfies all constraints in a TAMP problem. We refer the reader to [1] for a more detailed description.

There are several TAMP solvers that have been developed to use PDDLStream to define TAMP problems. One general approach is to optimistically assume that

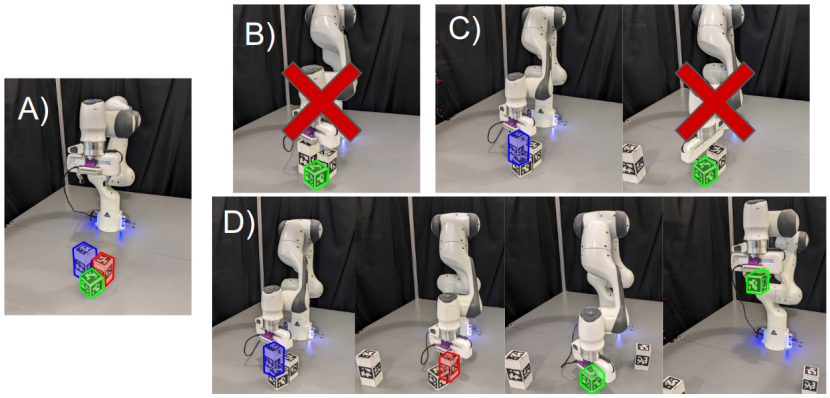

Fig. 1. A) A Panda agent is tasked with grasping the block highlighted in green. **B-C)** A naive abstract plan might be to grasp the green block directly, ignoring some or all potential obstructions. When the sub-problems associated with these plans are attempted, computation will be wasted solving for motion plans that are infeasible due to collision. **D)** Instead, our learned models guide planning by taking into account the feasibility and expected cost of sequences of sub-problems, and guide search to consider moving the obstructions first.

any time a stream is needed to certify an action predicate, it can be queried successfully, then generate abstract plans π, which contain actions that have unknown parameters as a result of this assumption. Using the approach outlined in [1], for a given abstract action plan π, we can generate the sequence of sub-problems s that must be solved to find the unknown parameters in π: referred to as the stream-plan ψ. If each $s \in \psi$ is able to be solved for satisfying output objects, then we can return an action plan that solves the original TAMP problem.

3 OUR APPROACH

Our objective is to find plans which solve TAMP problems efficiently, both in terms of wall-clock time spent searching for the plan, and with respect to the time it takes to execute the plan on a robot. We first describe how we construct and train simple models offline to predict the outcome and costs of individual sub-problems. Next, we demonstrate how to estimate and refine online the expected cost of a stream plan using these predictions. Finally, we show how this cost estimate is incorporated into a novel planner to more efficiently guide search.

3.1 Learning to Model External Functions

We train offline separate models for each sub-problem associated with a stream $s \in S$ in the domain, mapping an individual stream's inputs to estimates of the outcome and costs of querying for and executing the associated sub-problem. Each model has two types of input: local information that is defined in the domain of the stream (e.g., the start and goal configurations passed to a motion planner), and global information (e.g., the poses and dimensions of other objects in the scene). Global information, derived from a scene-graph, is embedded in a Graphical Neural Network. The GNN is composed of a node model, an edge

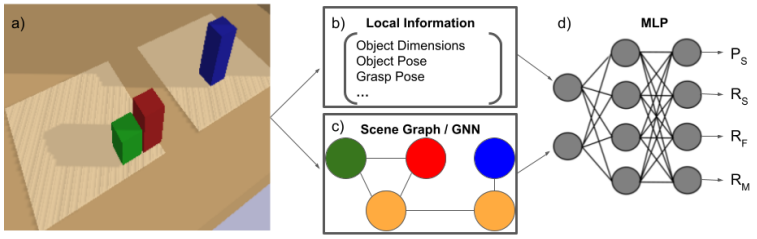

Fig. 2. An example of how we decompose a scene **(a)** into inputs for our various learned models. First we vectorize the local information that is relevant for a particular sub-problem **(b)**. Then, we embed the global information into a GNN **(c)**. After message passing, we concatenate the global feature vector of the graph with the local information and pass that through an MLP to generate the properties used to compute cost **(d)**.

model, and a global model, each of which shares the same architecture: two fully connected layers of size 128, with leaky ReLU activation after the first layer. For each object in our scene-graph, we pass the pose and dimension features through two fully connected layers, outputting a node feature vector that is N x 128. We similarly embed the edge features (the translation between two objects), drawing edges according to which objects are supported by the same surface, then perform one message passing step between the node and edge models. Next, we pass each node through the global model, and concatenate the softmax of the output with the vectorized local stream information. We pass this vector through a 4 layer MLP to produce the model's output. Training labels can be generated by tracking the outcome and costs each time a stream is queried during search. Figure 2 shows a representation of one such network.

Given local and global information, each network produces four outputs: P_S; the probability of solving the sub-problem for the given inputs, R_S; the expected cost of solving the sub-problem successfully in wallclock-time, R_F; the expected cost of failing to solve the sub-problem, and R_M; the expected time it would take to execute any motion plan output (if a sub-problem does not generate a trajectory, we set this value to 0). With these four properties, we have the ability to compute the expected cost of attempting to solve, then, if successful, executing any trajectory generated by a given sub-problem: $Q = P_S(R_S+R_M)+(1-P_S)R_F$

3.2 Evaluating Stream Plans using Learned Models

The above equation defines the expected cost of solving and executing a trajectory generated by one sub-problem. In order to evaluate the expected cost of a sequence of sub-problems ψ, we must consider the subsequent costs both in the case where we solve the sub-problem (we attempt to solve the next one in ψ), and when the attempt fails. In [7] and [8], the authors derived from the Bellman equation a method to estimate the cost of a sequence of stochastic actions with binary outcomes. Noting that the expected cost of execution R_M must be considered only for a complete plan, we recursively represent the expected planning cost Q_p of stream plan ψ, beginning at step t until the final step T, as:

$$Q_p(\psi_{t:T}) = P_{S_t}(R_{S_t} + Q_p(\psi_{t+1:T})) + (1 - P_{S_t})(R_{F_t} + Q_p(\psi_{0:t}) + Q_p(\psi_{t:T})), \quad (1)$$

where $Q_p(\psi_{t:T})$ represents the total cost (in seconds) it would take to solve each sub-problem $s_t \in \psi$ (but not yet execute any generated trajectories), beginning at step t in the plan. The cost of failing to solve a particular sub-problem must involve re-solving the sub-problems from step 0 up to that point in the plan to account for the fact that the different input parameters may be needed for a successful result. Manipulating Eq. 1 algebraically, we can re-write expected cost:

$$Q_p(\psi_{t:T}) = (R_{S_t} + Q_p(\psi_{t+1:T})) + \frac{1 - P_{S_t}}{P_{S_t}}(R_{F_t} + Q_p(\psi_{0:t})). \tag{2}$$

Due to the stochastic nature of some external solvers, certain sub-problem can be queried infinitely many times, and may eventually yield a successful output, particularly given different inputs. Eq. 2 represents this intuition, as the expected planning cost for a particular step in the recursion is simply the cost of solving a sub-problem, plus the number of times we expect to fail to solve s_t, times the expected cost of each failure. The ratio $\frac{1-P_{S_t}}{P_{S_t}}$ represents one less than the expected number of attempts before success in the geometric distribution parameterized by P_{S_t} (assuming independent samples). By rolling out the recursive steps, we can write the expected cost of successfully solving each sub-problem in ψ as follows:

$$Q_p(\psi_{t:T}) = \sum_{\tau \in t:T} \left(R_{S_\tau} + \frac{1 - P_{S_\tau}}{P_{S_\tau}}(R_{F_\tau} + Q_p(\psi_{0:\tau})) \right). \tag{3}$$

Notably, we have not yet considered the cost of executing the plan, as this cost is only relevant for complete plans, and so is not included in the cost of failure. Finally, we add the planning cost Q_p to the cost of executing all generated trajectories in a plan, Q_e, to recover the total cost of planning and execution Q:

$$Q(\psi_{t:T}) = Q_p(\psi_{t:T}) + \sum_{\tau \in t:T} R_{M_\tau} \tag{4}$$

Using Eq. (4), we are able to estimate the expected future cost of planning and execution from any point in a given ψ.

3.3 Planning with our Models

Even with our learned models, we cannot be certain of the feasibility and cost of a sequence of sub-problems without actually attempting to solve them. To account for this uncertainty, we formulate TAMP as a stochastic search problem. In *eTAMP* [2], the authors propose using Progressive Widening for Upper Confidence Bounded Trees (PW-UCT) [9, 10] to search for parameters that satisfy the constraints of an action plan. We build upon this approach for our planning algorithm, using our models to guide search, and so outline it briefly here.

Searching for Abstract Stream Plans: The first step in our approach is to generate several abstract action plans $\pi \in \Pi$ — sequences of actions which would represent a successful plan if all preconditions are met — from a PDDLStream problem using a top-k planner [11]. As mentioned in Sec. 2, some or all of an action's preconditions may be certifiable only by a stream. However, during the search for abstract action plans, we do not explicitly attempt to solve the sub-problems associated with those streams, as this would be prohibitively expensive.

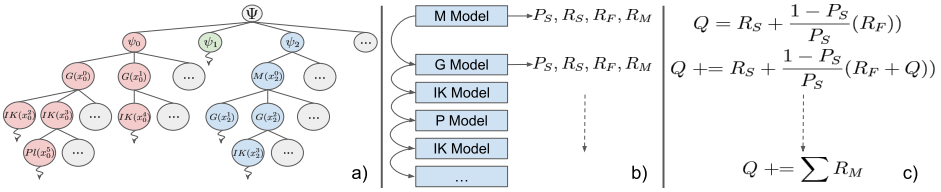

Fig. 3. A) The root node in our tree represents a set of sub-problem sequences Ψ, and each child is a sample ψ from that set. For all subsequent nodes, the available actions consist of attempting to solve a sub-problem, in this case: G: sample grasp, Pl: sample stable placement, IK: solve for kinematically feasible configuration, and M: solve for collision free motion plan. **B)** At each node, we use learned models to predict the feasibility and costs of attempting to solve all remaining sub-problems in the sub-tree. **C)** We use these predictions to estimate the expected cost of finding satisfying solutions for the remaining steps in ψ from each node using eq. (4), defined in Sec. 3.2.

Instead we solve for abstract plans π, and compute the associated stream-plan ψ for each. Given a set of k stream plans $\psi \in \Psi$, we can begin to attempt to search for these parameters.

One approach for this search would be to simply query Equation (4) for each stream plan given the initial state of the problem, and guide search using these estimates only. However, during search in non-trivial domains, we may need to consider different potential solutions to the same sub-problem (e.g., sample multiple different grasps on a block). Because the output of a sub-problem may depend on its input, there can and will be different cost and feasibility estimates for the same step in a stream-plan depending on the parameters that are passed to the model (which depend on the solutions to upstream sub-problems). Therefore, when a new solution to a sub-problem is found, we must update our predictions for the remainder of a stream plan. As we progress in our search for the parameters of a plan, some predicted values will vary, and so too will the remaining estimated expected cost. The differences in estimates can help us guide search, and we account for the associated uncertainty with PW-UCT.

Searching within Stream Plans: There are four distinct steps in a PW-UCT search problem. The first is *selection*, where the existing tree is navigated according to the UCT heuristic (Eq. (5)) to find a node to expand. Next, in the *expansion* phase, a child from the selected node is generated and appended to the tree. In the *simulation* step, we continually add nodes from the newly expanded child until either an expansion fails, or we successfully reach our goal. Finally, we update the statistics (total node visits and accrued reward) of all visited nodes via *back-propagation*. This process is repeated until a solution is found.

Our tree is built as follows. At the root node (level 0), the available actions correspond to selecting one of the stream plans returned by the top-k planner. After this choice, each level-1 subtree is associated with different evaluations for that stream plan. Each level in a sub-tree corresponds to a specific stream, and each node in a level to a solution of the associated sub-problem. Because sub-problems re-solved and potentially produce novel results, there are infinitely many actions from each node (although a tree will only get as deep as the length of a stream plan). Refer to Figure 3 for a depiction of a growing search tree.

Guiding Search with Learned Cost: During traversal from the root to a leaf in the selection phase, the UCT equation (5) is used to choose the next node:

$$argmax_{v_i} Q(v_i) + c\sqrt{\frac{2ln(N(v))}{N(v_i)+1}}, \tag{5}$$

where v_i represents a child of node v, $N(v)$ denotes the number of times a node has been visited, and $Q(v_i)$ gives the online estimated value of a particular child.

The UCT equation (5) relies on an estimate of Q in order to guide search. In *eTAMP*, the authors propose a heuristic based on the depth of the search tree, and accrued reward [2]. Such heuristics, while potentially useful, require domain knowledge, may necessitate tuning, and cannot adapt to different streams within a plan. Instead, we use our learned models, applied to each step of the remaining stream plan, to more efficiently estimate Q and guide search. We evaluate Eq. (4) for each visited node in order to get an estimate for remaining expected cost. Specifically, we use the negative of the final output from Eq. (4) as the estimate for Q. If, in the application of Eq. (4), we encounter a stream input that has not yet been solved for, we pass the model a zero-vector of the same shape (along with a flag in the input) in its place to make a prediction without that information, and remove any associated edges in the scene-graph GNN embedding.

If the node selected by our learned UCT estimation is a leaf node (meaning it has no children), the associated sub-problem is attempted, and, if solved, a new node is added to the tree. Then, we re-query our learned models for the remaining sequence of streams in ψ, using any new output generated by solving the previous sub-problem, and compute the learned Q-value for the new node given those new inputs. From there, the simulation and back-propagation steps are taken, and selection begins again. By growing the tree in this way, we are biased to evaluate sub-problems that are determined by our learned cost to be the most likely to lead to a satisfying plan in the shortest amount of time, balancing exploiting high value branches, and exploring new solutions to account for potential inaccuracies in the learning. We are also able to consider the act of sampling a new node as another action in Eq. (5). If the UCT heuristic for re-solving a particular sub-problem —according to the estimated cost from Eq. (4)— is higher than that of any the available children, we do so, and add a new child to the current node. Once a full sequence of sub-problems has been successfully solved, we have solved the original TAMP problem, and can return the full action-plan.

4 Experimental Results

To highlight the capabilities of our learned planner, we implement our approach in two simulated scenarios, as well as on a real robot. To make comparison straightforward, the simulated experiments use two platforms/environments tested in *eTAMP*; specifically, their 'kitchen' and 'unpack' domains [2]. We compare our planner against the heuristic-driven planner defined in *eTAMP*, using the hand-defined parameters, tuned for each environment as specified by the authors. In each instance, we demonstrate that our planner is able to out-perform the baseline [2] in both planning time and the number of search nodes expanded, while finding plans of equivalent motion cost as shown in Table 1.

Table 1. Experimental Results: All units in seconds (% cost reduction)

	Kitchen Domain		Unpack Domain		Real Domain	
Metric ↓ (↑)	**Baseline**	**Ours**	**Baseline**	**Ours**	**Baseline**	**Ours**
Planning Cost	98.3	39.5 (60%)	105.8	54.3 (49%)	66.7	26.2 (61%)
Motion Cost	86.4	82.1 (5%)	46.7	46.0 (1%)	21.3	20.9 (1%)
Total Cost	184.6	121.7 (34%)	152.6	100.4 (34%)	88.0	47.1 (46%)
Node Expansions	149.8	55.2 (63%)	707.2	125.9 (82%)	91.2	11.0 (88%)

4.1 Kitchen Domain

In the 'kitchen' domain, the robot is a simulated PR2, shown in figure 4 with five available actions: pick up an object, place an object on a platform, move between configurations, cook an object, and clean an object. An object is cleaned when placed on the sink and cooked when placed on the stove. The agent is then tasked to first 'clean', then 'cook' all blocks initially placed on a table, being careful to avoid overcrowding any platform. We consider four distinct sub-problems, specifically a grasp sampler, a stable placement sampler, an inverse kinematics solver, and an RRT motion-planner between configurations. We refer the reader to [2] for a more detailed description of the domain.

For this task, most action plans considered by the planner are feasible, though deciding which sub-tree to explore in search is difficult. Notably, the order in which blocks are moved is generally irrelevant to the feasibility and cost of the problem. Therefore, at the root node, our learned models predict approximately the same cost for each possible sequence of sub-problems ψ. However, as the search tree grows in depth, and blocks are added to the final platform, we are able to improve upon naive search by considering how cluttered the surface is, and if a particular sampled pose is feasible.

In this domain, we ran 1600 trials for both the baseline planner [2] and our learned planner (trained on 100 trials worth of data collected by running the baseline planner), initializing each trial with a new random seed. We recorded both the planning and execution time, and plotted these values for the individual trials (along with their sum) in Fig. 4 and Table 1. As shown in the table, we demonstrate a mean reduction in total time of approximately 63 seconds (a 34% improvement). We demonstrate that our approach can find plans of similar execution time to the baseline with fewer average node expansions (150 vs 55).

4.2 Unpack Domain

In our second environment, we consider a table-top manipulator with the ability to pick and place objects on different platforms. As before, to pick or place an object our agent must sample grasps or placement poses, find collision-free configurations, and compute safe trajectories between these configurations. In this domain, our goal is simply to move a specified object from one platform to the other. However, depending on the configuration of the other objects in the scene, this may not be immediately feasible. Whereas in the 'kitchen' domain, nearly every high-level plan could be valid depending on the object groundings, in the 'unpack' domain, many of the abstract action plans returned by the top-k planner are infeasible depending on the orientation of the blocks in the scene. For example, if the taller blue block sits beside the green block, the planner will

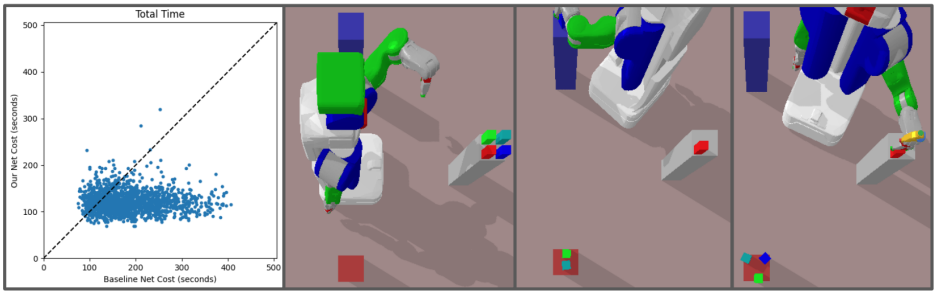

Fig. 4. A comparison between the total cost of planning and execution of the baseline planner [2] and our learned TAMP planner for 1600 trials in the 'kitchen' domain. Each point in the scatter-plot represents the outcome of a single trial. The images on the right demonstrate potential failure modes in this domain. If the first few blocks are placed poorly on the stove, it may be impossible to safely place all four blocks there without risking collision. Our approach allows us to predict when a block placement will lead to failure later in a plan, reducing the time spent planning in these sub-trees.

be unable to find a configuration to grasp the green block without coming into collision with the blue one. As such, any calls to our IK solver will fail, and the ability to reason about which queries to to an external planner will or will not succeed can be very impactful in terms of accelerating planning. We highlight a few examples of this in Fig. 5.

During training, over 200 trials, we considered instances with either one, two, or three blocks in the scene, with the initial poses of the blocks randomly selected (but closely clustered). We evaluated our planner for the case of three blocks, running 400 trials for both the baseline planner [2] and our learned planner. Once again, we recorded both the planning and execution time, and plotted these values for the individual trials in Fig. 5 and Table 1.

We further compare our approach to one (not included in the table) which uses predicted feasibility as a threshold to prevent the planner from exploring low-probability actions [12]. In that work, the authors report an improvement of 63% over the same baseline in terms of motion planning time only in the 'unpack' domain, which does not include time querying the top-k solver for Ψ. Using our approach, we found a savings of approximately 67% in this metric. Moreover, because this approach thresholds certain sub-trees from ever being considered, the planner fails to find any plan $\sim 8\%$ of the time, whereas we found no failures over 400 trials. We do note that we did not re-implement and test this approach in this work, and are relying on the reported values.

4.3 Real World Experiments

Finally, we implement our planner on a real Panda manipulator (see Fig. 1), testing a modified version of the 'unpack' problem, where the robot is tasked with grasping a particular block in a crowded grouping, potentially having to remove obstructions before it can. The Panda arm is equipped with a parallel gripper, and a RealSense camera, which it uses to identify objects, their positions, and their shapes. We define grasp and placement samplers, as well as an RRT motion

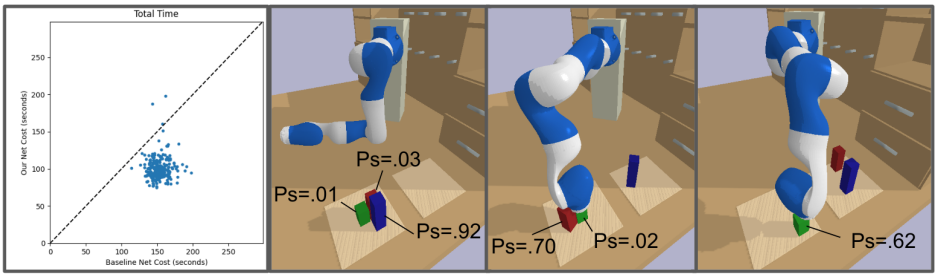

Fig. 5. Comparing the total cost of planning and execution for 400 trials in the 'unpack' domain, where once again each point in the scatter-plot represents the outcome of a single trial. We also highlight some example scenes, where we show the feasibility predictions given by our IK model for attempting to grasp each block on the first platform. As more blocks are cleared, the predicted likelihood that we can successful find a collision-free configuration to grasp the green block increases.

planner, and an IK solver as the relevant sub-problems. We train our models for each sub-problem on data from 100 simulated trials, then test on the real robot.

Over 10 real world trials, we compare our planner to the baseline approach for identical initial conditions. For each trial, first the robot identifies the blocks in the scene and their poses. Then, we search for a plan to grasp the selected block using our approach. Finally the agent executes this plan, using the mounted camera to account for perception errors during execution as needed. In each trial, our planner outperforms the baseline with respect to planning time and nodes expanded, while producing plans that are of equivalent quality in terms of motion cost. An example of this scenario is shown in Fig. 1, and we report the results of the trials in Table 1.

5 Related Work

Viewing TAMP as a hybrid constraint satisfaction problem [6], most TAMP approaches can be categorized as solving the constraints jointly or individually. In the first approach, the problem is written as one large constrained optimization problem (typically a Mixed Integer Program), where discrete components such as which block to pick up are represented by integers, and the trajectory optimization is real-valued [6, 13, 14]. These joint optimization strategies can be appealing, as satisfying a single optimization solves the entire problem. However, such approaches are often limited in that certain aspects of the problem may not be easily differentiable, efficiently re-using computation can be difficult, and it might not always be straight-forward to incorporate off-the-shelf samplers/solvers [6]. The primary alternative approach is to consider solving for sets of parameters that satisfy small groups of constraints, and combine the solutions into actions and plans. For example, we can sample a block placement on a platform that is free of collisions, then confirm it is positioned so that there exists a kinematically feasible configuration to execute such a placement. Approaches that break up the problem in this way can take advantage of external tools that are optimized for specific sub-problems (e.g, Fast Downward for the discrete task problem, efficient inverse-kinematics solvers, or neural networks for grasp sampling) [1, 2, 6]. In this work, we build upon this second approach.

There has been significant recent progress in improving planning for TAMP problems using learned models. Most relevant to this work are those contributions which attempt to accelerate search from experience [12, 15–21]. Some learn explicitly which components of a given domain are relevant for a particular TAMP problem, though do not further guide search within their reduced domain [17, 18]. Kim et al., learn an action sampling distribution for geometric motion planning problems, but do not take advantage of off-the-shelf samplers/solvers [19]. Kim and Shimanuki learn a Q-function as a heuristic to use in search for a geometric TAMP problems [16], however do not learn to bind the continuous parameters of its actions. Closely related, Khodeir et al., [21] specifically scores the relevance of streams within the PDDLStream framework, and improve search for stream-plans. However, they do not consider the search for an action's parameters. An interesting direction for future work would be to combine the two approaches to improve both the search for stream plans, and the parameters thereof. Finally, the work in [12] learns a feasibility predictor from images to accelerate *eTAMP* [2]. However, this work does not predict costs, and thresholds the predicted feasibility to bound branches in search. As mentioned in section 4, this can lead to planning failure if a feasible branch is below the defined threshold.

Outside of TAMP, there has been work in planning with learned outcomes. Specifically, Stein et al., learn models to predict the success/failure and costs of actions for planning in long-horizon, partially observable domains, however are limited to navigation tasks [7, 8]. We build upon the intuition of this work in sec. 3.2. Xu et al., propose simultaneously learning outcomes and action dynamics, and plan in a learned latent space [22]. However, they cannot utilize optimized off-the-shelf planners to solve TAMP problems with their learned representation.

6 Discussion

In this work, we demonstrated that our strategy for learning models to predict the feasibility and costs of relevant sub-problems to guide TAMP is effective in various problem domains. We achieved improved performance with respect to planning time across three domains, including one on a real Panda robot. Furthermore, our approach has the potential to be widely applicable, as it is compatible with different learning methods, whether the inputs are simple vectorized object properties (as they are here), visual information, or even language.

One possible application of our approach is in domains where there is noise or partial observability in the world model. In such environments, it may not be efficient to query streams during initial planning. For example, if the pose of a far away object is uncertain, attempting to solve for the exact configuration for a grasp is likely a waste of computation. If instead we can predict that we will be able to find such a configuration once the pose is more certain, we can wait to query an IK solver until the robot is near the object. This is left for future work.

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*, 2020.

2. T. Ren, G. Chalvatzaki, and J. Peters, "Extended tree search for robot task and motion planning," *arXiv preprint arXiv:2103.05456*, 2021.
3. M. Ghallab, D. Nau, and P. Traverso, *Automated planning and acting*. Cambridge University Press, 2016.
4. E. Karpas and D. Magazzeni, "Automated planning for robotics," *Annual Review of Control, Robotics, and Autonomous Systems*, 2020.
5. S. M. LaValle, *Planning algorithms*. Cambridge university press, 2006.
6. 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*, 2021.
7. G. J. Stein, C. Bradley, and N. Roy, "Learning over subgoals for efficient navigation of structured, unknown environments," in *CoRL*. PMLR, 2018.
8. C. Bradley, A. Pacheck, G. J. Stein, S. Castro, H. Kress-Gazit, and N. Roy, "Learning and planning for temporally extended tasks in unknown environments," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021.
9. G. M. J. Chaslot, M. H. Winands, H. J. v. d. Herik, J. W. Uiterwijk, and B. Bouzy, "Progressive strategies for monte-carlo tree search," *New Mathematics and Natural Computation*, 2008.
10. R. Coulom, "Computing "elo ratings" of move patterns in the game of go," *ICGA journal*, 2007.
11. D. Speck, R. Mattmüller, and B. Nebel, "Symbolic top-k planning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2020.
12. 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.
13. M. A. Toussaint, K. R. Allen, K. A. Smith, and J. B. Tenenbaum, "Differentiable physics and stable modes for tool-use and manipulation planning," 2018.
14. E. Fernandez-Gonzalez, E. Karpas, and B. Williams, "Mixed discrete-continuous planning with convex optimization," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2017.
15. D. Driess, J.-S. Ha, R. Tedrake, and M. Toussaint, "Learning geometric reasoning and control for long-horizon tasks from visual input," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021.
16. B. Kim and L. Shimanuki, "Learning value functions with relational state representations for guiding task-and-motion planning," in *CoRL*. PMLR, 2020.
17. T. Silver, R. Chitnis, A. Curtis, J. B. Tenenbaum, T. Lozano-Perez, 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*, 2021.
18. C. Agia, K. M. Jatavallabhula, M. Khodeir, O. Miksik, V. Vineet, M. Mukadam, L. Paull, and F. Shkurti, "Taskography: Evaluating robot task planning over large 3d scene graphs," in *CoRL*. PMLR, 2022.
19. B. Kim, L. Kaelbling, and T. Lozano-Pérez, "Guiding search in continuous state-action spaces by learning an action sampler from off-target search experience," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018.
20. T. Pan, A. M. Wells, R. Shome, and L. E. Kavraki, "Failure is an option: Task and motion planning with failing executions," in *2022 International Conference on Robotics and Automation (ICRA)*. IEEE, 2022.
21. M. Khodeir, B. Agro, and F. Shkurti, "Learning to search in task and motion planning with streams," *IEEE Robotics and Automation Letters*, 2023.
22. D. Xu, A. Mandlekar, R. Martín-Martín, Y. Zhu, S. Savarese, and L. Fei-Fei, "Deep affordance foresight: Planning through what can be done in the future," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021.