2024 IEEE International Conference on Robotics and Automation (ICRA)
May 13-17, 2024. Yokohama, Japan

COAST: COnstraints And STreams for Task and Motion Planning

Brandon Vu, Toki Migimatsu, Jeannette Bohg

2024 IEEE International Conference on Robotics and Automation (ICRA) | 979-8-3503-8457-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICRA57147.2024.10611670

***Abstract*— Task and Motion Planning (TAMP) algorithms solve long-horizon robotics tasks by integrating task planning with motion planning; the task planner proposes a sequence of actions towards a goal state and the motion planner verifies whether this action sequence is geometrically feasible for the robot. However, state-of-the-art TAMP algorithms do not scale well with the difficulty of the task and require an impractical amount of time to solve relatively small problems. We propose Constraints and Streams for Task and Motion Planning (COAST), a probabilistically-complete, sampling-based TAMP algorithm that combines stream-based motion planning with an efficient, constrained task planning strategy. We validate COAST on three challenging TAMP domains and demonstrate that our method outperforms baselines in terms of cumulative task planning time by an order of magnitude. You can find more supplementary materials on our project website.**

I. Introduction

We aim to equip a robot with the ability to solve complex long-horizon tasks that require a combination of symbolic and geometric reasoning. *Task and Motion Planning* (TAMP) is an approach for solving such tasks. TAMP methods often use task planning to produce a sequence of symbolic actions, i.e. a task plan, in addition to using sampling-based motion planning to ensure the task plan is geometrically feasible. If the task plan is geometrically infeasible, then this result needs to be communicated to the task planner for replanning. Two main paradigms of communication exist: *sample-first* and *plan-first* [1]. *Sample-first* methods perform motion sampling first (without any task plan) and then query task planning to sequence only the geometrically feasible samples [2, 3]. *Plan-first* methods perform task planning first and then refine the task plans with motion sampling, where sampling failures due to geometric infeasibility are translated into task planning constraints for the next iteration of task planning [4–6].

Two sampling-based TAMP algorithms closely related to our work are PDDLStream [3] and Iteratively Deepened Task and Motion Planning (IDTMP) [4]. PDDLStream is an optimistic *sample-first* method that breaks down motion planning into black-box sampling functions called streams and integrates them into the task-planning problem as objects and action preconditions in the *Planning Domain Definition Language* (PDDL) [7]. A limitation of PDDLStream is that it must generate the symbolic objects to be used for task planning without knowing which ones may be necessary. But generating too many objects results in exponential task planning times. IDTMP is a *plan-first* method that treats task planning as Constraint Satisfaction Problems (CSPs) and uses constraints to communicate motion planning failures. A limitation of IDTMP is that it requires a discretization of the workspace which prevents IDTMP from applying to domains with large workspaces like our *Kitchen* and *Rover* domains.

The authors are with the Department of Computer Science, Stanford University (e-mail: {vubt, takatoki, bohg}@cs.stanford.edu).

Toyota Research Institute provided funds to support this work.

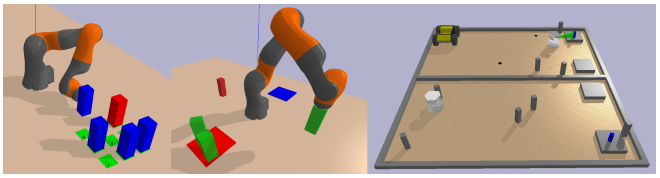

Fig. 1: We propose COAST, a sampling-based TAMP algorithm that is able to solve complex, geometrically constrained, long-horizon planning problems faster than prior state-of-the-art. We demonstrate the ability of our algorithm to solve problems from three domains: a 3×3 grid rearranging task (*Blocks* [4], *left*), a constrained pick-and-place kitchen task (*Kitchen* [3], *middle*), a rover surveillance task with obstacles (*Rover* [3], *right*).

We propose a probabilistically-complete, *plan-first* TAMP algorithm that is significantly faster than PDDLStream and IDTMP. This speedup occurs by using a direct stream planning algorithm to create stream objects *after* task planning rather than *before* to avoid the computational cost of task planning with many unnecessary stream objects. We validate our method on three TAMP domains (Fig. 1), each one challenging in different ways, and demonstrate that our method outperforms both PDDLStream and IDTMP by an order of magnitude in task planning time.

II. Related Works

TAMP problems are challenging due to the need to search over both discrete and continuous spaces. Thus, many works propose different techniques to reduce the search complexity of TAMP. Hierarchical Task Networks (HTN) [8] are a class of TAMP algorithms that use expert-designed hierarchies to reduce the dimension of the search problem [9]. Other works [10–12] extend this approach for manipulation. Instead of using hierarchies, Srivastava et al. [13] is a plan-first framework that uses constraints defined in PDDL to prune infeasible plans. We also use constraints and a plan-first approach, but unlike this prior work, our method is applicable outside of manipulation and does not assume the absence of reachable dead-end states. We accomplish this by using streams as an abstraction for motion planning while also using queue-based algorithms that revisit previous states, relaxing the dead-end state assumption. Our focus lies in increasing the speed of universally applicable TAMP algorithms, agnostic to the motion planning implementation. To achieve this, we advance the integration of classical task planning [14] in PDDL [7] and stream-based motion planning [3].

The motion planning component of TAMP commonly

consists of finding valid robot trajectories for manipulating objects while avoiding collisions in addition to finding satisfiable geometric assignments such as grasps and poses [15]. Optimization-based approaches [16, 17] attempt to find optimal motion plans with nonlinear optimization, which can be sensitive to initial conditions and prone to failure. Thomason et al. propose TMIT* [5] which plans in a hybrid symbolic and continuous state space, using CSP constraints and asymmetric forward and reverse motion planning. Our framework provides a connection from constraint-based planning to streams, which could be extended to an approach like TMIT*.

In contrast to refining each symbolic action sequentially, recent works propose to break down each motion planning sub-problem into even smaller, reusable functions that enable more efficient motion planning across actions. PDDLStream [3] proposes a general TAMP framework that formalizes these as *streams*. Decomposing the motion planning problem into these lightweight samplers gives rise to efficient sampling algorithms that can intelligently resample streams until a task plan is refined. However, incorporating streams directly into task planning causes the PDDL problem to exponentially grow in the number of objects, which slows down task planning. We propose a stream-based TAMP algorithm that benefits from efficient motion planning while keeping task planning light.

There has been a rise in learning-based methods that seek to overcome the disadvantages of classical TAMP. Driess et al. [18] trains a policy to solve TAMP tasks from images and demonstrations produced by classical TAMP solvers. Other works augment classical TAMP solvers by accelerating planning with learned heuristics [19–21] or giving them the ability to handle uncertainty [22, 23]. Our work can benefit these methods by speeding up solving times and increasing the scale of possible tasks.

PDDLStream [3] and IDTMP [4] are the two works most closely related to ours. We build off of PDDLStream's stream framework for motion planning and propose a constrained task-planning method similar to that of IDTMP.

III. Background

A. Planning Domain Description Language

Our framework uses the *Planning Domain Definition Language* (PDDL) [7] for task planning. A PDDL domain, typically defined as a `domain.pddl` file, can be described as a tuple $(\Phi, \mathcal{A})$, where Φ is the set of predicates (binary-valued properties of objects) and $\mathcal{A}$ is the set of actions. A PDDL problem, typically defined as a `problem.pddl` file, is a tuple $(\mathcal{O}, s_0, g)$, where $\mathcal{O}$ is the set of environment objects, s_0 is the initial state, and g is the goal formula to be satisfied. The task planner's role is to find a sequence of actions, or a task plan π, that will turn the initial state s_0 into a new state that satisfies the goal formula g.

Actions are defined by their preconditions—a logic formula that must be satisfied by the current state in order to execute the action—and effects—a formula that describes how the state changes upon executing the action. The following are PDDL definitions of `Pick` and `Place` actions for a simple pick-and-place domain that we will use as a running example in this paper.

```
(:action Pick
  :parameters (?o - obj ?r - region)
  :precondition (and
    (on ?o ?r)
    (handempty))
  :effect (and
    (not (on ?o ?r))
    (holding ?o)
    (not (handempty))))
(:action Place
  :parameters (?o - obj ?r - region)
  :precondition (holding ?o)
  :effect (and
    (on ?o ?r)
    (not (holding ?o))
    (handempty)))
```

B. Streams

Our framework uses streams from PDDLStream [3] to perform sampling-based motion planning. Streams decompose the long-horizon motion planning problem into unit sampling functions that each address a small component of the motion planning problem.

A stream is defined as a generator function $\sigma(x_1, \ldots, x_n) \rightarrow (y_1, \ldots y_m)$ which takes a tuple $x_1, \ldots, x_n$ as input and generates a tuple of outputs $(y_1, \ldots, y_m)$. For example, the `sample-pose` stream may take in an object o and region r and output a placement pose p for o in r:

```
(:stream sample-pose
  :inputs (?o - obj ?r - region)
  :outputs (?p - pose))
```

Streams are typically defined in a `streams.pddl` file and are accompanied by user-provided Python functions which implement the actual sampling procedure. In this paper, we refer to the set of streams in the PDDLStream domain as Σ. We extend streams by adding a cache and a probability to return a cached result with a probability from $[0, 1)$. We only use this feature for the *Rover* domain and ablate it in Sec. V.

A stream *instance* is a stream instantiated with concrete inputs and outputs, such as `sample-pose(apple, shelf)→p1`. Outputs of stream instances are called stream objects, and the inputs to stream instances can either be stream objects or PDDL objects. In this case, `apple` and `shelf` are both PDDL objects, while `p1` is a stream object. PDDL objects are defined in the PDDL problem as object set $\mathcal{O}$, but stream objects do not exist at the beginning of task planning and need to be created during planning.

Every time a stream instance is called, it may generate new values for the output stream objects. For example, `sample-pose` might generate poses for `p1` where the positions are sampled randomly from the support area of `shelf`. A stream instance also returns a "certified fact", or a symbolic proposition, along with the sampled values to certify the success of sampling. For example, if `sample-pose` succeeds, it would output the certified fact `(sample-pose apple shelf p1)`, which indicates that `p1` is a valid pose for `apple` on `shelf`. However, if sampling fails—for example,

because `shelf` is too small to support a placement pose for `apple`—then the certified fact is not returned.

Note that our definition of certified facts is slightly different from that of PDDLStream; while stream instances in PDDLStream may output multiple certified facts, we require that stream instances output a single certified fact containing all of the input and output objects so that there is a bijective mapping between stream instances and certified facts. This is not a restriction because non-bijective certified facts are instead added to the geometric postconditions of our action. The bijective mapping allows our stream planner to directly determine the required stream instances from a set of desired certified facts. PDDLStream, on the other hand, must blindly create stream instances until all desired certified facts are covered, which is an expensive iterative process and often results in the creation of many unused certified facts in the task state.

A stream plan ψ is a sequence of stream instances that each must be sampled successfully to complete the motion planning problem for a candidate task plan π. After computing a stream plan for a candidate action skeleton, the last step is to sample the streams to generate a motion plan. If a stream fails, we resample the streams until the entire stream plan is successful. We also need to decide when to give up motion planning for a candidate action skeleton and mark it as infeasible. In our experiments, we use the semi-complete `Adaptive` PDDLStream algorithm to handle stream plan sampling and termination. We refer readers to [3] for an in-depth description of this algorithm.

The key difference between PDDLStream and our method is how stream objects and certified facts are treated. PDDLStream treats stream objects as PDDL objects and allows certified facts to be used in the preconditions of PDDL actions. The main disadvantage of this approach is that the task planner is now required to decide what stream objects to use for a task plan. It is not known a priori what stream objects are required to satisfy the task planning goal. Therefore, stream generation in PDDLStream is an iterative process where stream objects are incrementally introduced by level to the PDDL problem until the task planner succeeds. At first, when no stream objects are available, the task planner will certainly fail. As the number of stream objects grows, task planning quickly becomes intractable due to its PSPACE-hard complexity. Task planning is therefore a significant bottleneck in PDDLStream when many stream objects are required, which may happen for problems that have many movable objects and require long task plans. Our key insight is that deciding what stream objects to use for a task plan can be done with a simple stream planning procedure (Sec. IV-A) that does not require a PDDL solver. The integration between task and motion planning is achieved with PDDL constraints (Sec. IV-B) rather than deferring the stream instance to a later level like in PDDLStream.

IV. COAST Algorithm

The PDDL domain $(\Phi, \mathcal{A})$ and problem $(\mathcal{O}, s_0, g)$ given to COAST are defined with vanilla PDDL (i.e. no stream objects), with actions resembling the example definitions of `Pick` and `Place` in Sec. III-A. To connect the PDDL domain with streams Σ, we introduce a stream planning layer that plans using geometric actions $\mathcal{A}_{\text{geom}}$ and the initial geometric state $s_{0_{\text{geom}}}$, explained in more detail in Sec. IV-A. Then, COAST enters a loop that alternates between task and motion planning. An overview of our algorithm is provided in Alg. 1. We use an off-the-shelf PDDL solver to propose a candidate task plan π that satisfies the symbolic goal but is not necessarily valid from a motion planning standpoint (Line 6). We then run a novel stream planning method to ground the task plan π with stream objects to produce a geometric task plan π_{geom} (Line 9). For example, if π is the task plan [`Pick(apple, table)`, `Place(apple, rack)`], then π_{geom} might look like [`Pick(apple, table; g1)`, `Place(apple , rack; g1, p1)`], where `Pick` and `Place` are grounded with stream objects necessary for motion planning, such as grasp `g1` and pose `p1`. Stream planning also produces a stream plan ψ, which is a sequence of stream instances that need to be sampled to produce values for the stream objects in π_{geom}. We then use PDDLStream's `Adaptive` algorithm to sample the stream plan and return a binding map Y from the stream outputs y to their sampled values along with the set of certified facts s_ψ (Line 10). If there is one certified fact per stream instance in ψ, then the entire stream plan was sampled successfully and we can terminate planning. Otherwise, we constrain the PDDL problem by modifying the initial state s_0 and action definitions in $\mathcal{A}$ to force the PDDL solver to find an alternative plan (Line 13). Then the planning cycle continues until we successfully complete motion planning for a task plan or time out. We maintain a queue of all previous planning states, and we sort the queue by the frequency of the set of constraints and the number of constraints to prioritize more unique and less constrained task states. We prove our algorithm is probabilistically complete in Sec. VII. The following subsections describe our stream planning procedure and task planning constraints in more detail.

Algorithm 1 COAST TAMP Algorithm Overview

1: **function** COAST($\Phi, \mathcal{A}, \mathcal{A}_{\text{geom}}, \Sigma, \mathcal{O}, s_0, s_{0_{\text{geom}}}, g$)
2: $\quad\mathcal{Q} \leftarrow []$
3: $\quad$PUSH($\mathcal{Q}, \Phi, \mathcal{A}, \mathcal{O}, s_0, g$)
4: $\quad$**while** not TIMEOUT **do**
5: $\quad\quad$POP($\mathcal{Q}, \Phi, \mathcal{A}, \mathcal{O}, s_0, g$)
6: $\quad\quad\pi \leftarrow$ TASKPLAN($\Phi, \mathcal{A}, \mathcal{O}, s_0, g$)
7: $\quad\quad$**if** $\pi =$ None **and** $Q = []$ **then**
8: $\quad\quad\quad$**return** None
9: $\quad\quad\pi_{\text{geom}}, \psi \leftarrow$ STREAMPLAN($\mathcal{A}_{\text{geom}}, \Sigma, s_{0_{\text{geom}}}, \pi$)
10: $\quad\quad Y, s_\psi \leftarrow$ ADAPTIVEBINDING(ψ)
11: $\quad\quad$**if** ISSUCCESSFUL(ψ, s_ψ) **then return** π_{geom}, Y
12: $\quad\quad$PUSH($\mathcal{Q}, \Phi, \mathcal{A}, \mathcal{O}, s_0, g$)
13: $\quad\quad s_0, \mathcal{A} \leftarrow$ CONSTRAINPDDL($\mathcal{A}, \psi, s_\psi, s_0$)
14: $\quad\quad$PUSH($\mathcal{Q}, \Phi, \mathcal{A}, \mathcal{O}, s_0, g$)

A. COAST Stream Planning

For every candidate task plan produced by the task planner, we need to perform motion planning to produce

Algorithm 2 COAST Stream Planning

function STREAMPLAN($\mathcal{A}_{\text{geom}}, \Sigma, s_{0_{\text{geom}}}, \pi$)
  $\psi \leftarrow \varnothing$
  for $t = 1 \ldots H$ **do**
    $a_t \leftarrow \pi[t]$
    $a_{t_{\text{geom}}} \leftarrow$ GETGEOMACTION($\mathcal{A}_{\text{geom}}, a_t$)
    $\overline{a}_{t_{\text{geom}}} \leftarrow$ GROUNDGEOMACTION($a_{t_{\text{geom}}}, s_{t-1_{\text{geom}}}$)
    $\psi \leftarrow \psi \cup$ GETPRECONDITIONSTREAMS($\overline{a}_{t_{geom}}$)
    $s_{t_{\text{geom}}} \leftarrow$ APPLYGEOMACTION($s_{t-1_{\text{geom}}}, \overline{a}_{t_{\text{geom}}}$)
  return ψ

a trajectory for the robot to execute the task plan, or if motion planning fails, mark the task plan as infeasible. We define the motion planning problem as finding satisfiable assignments to stream objects in a given stream plan. Our method uses a novel stream planning subroutine to generate a stream plan from a candidate task plan. The pseudocode for this subroutine is provided in Alg. 2.

Each action is associated with a set of streams that need to be executed during motion planning. We specify how these streams are executed for each action in a separate `geometric.pddl` file. For example, we may define the geometric `Place` action as:

```
(:geom-action Place
  :parameters (?o - obj ?r - region)
  :inputs (?g - grasp)
  :outputs (?p - pose)
  :geom-precondition (and
    (in-grasp ?o ?g)
    (sample-pose ?o ?r ?p))
  :geom-effect (and
    (not (in-grasp ?o ?g))
    (at-pose ?o ?p)))
```

The **`:parameters`** field defines the PDDL object parameters for this action; it should be equivalent to **`:parameters`** defined for the corresponding PDDL action in `domain.pddl`. The **`:inputs`** and **`:outputs`** fields define the input and output stream objects for this action. This `Place` action, for example, takes as input a grasp `?g` and outputs a pose `?p`. While the **`:parameters`** will be determined by the task planner (e.g. `Pick(apple, table)`), the stream objects need to be determined during the stream planning phase.

During stream planning, we maintain a geometric state, which, similarly to the symbolic state in PDDL, is represented with a set of geometric propositions. While symbolic propositions like `(on apple table)` are defined with symbolic objects, geometric propositions can also be defined with stream objects, such as `(at-pose apple p1)`. The **`:geom-precondition`** field defines the requirements for applying a geometric action to a geometric state, and the **`:geom-effect`** field specifies how the geometric state changes upon executing each action—just like the preconditions and effects of symbolic actions in PDDL.

The job of stream planning is two-fold: 1) grounding each action in a given task plan with stream objects, and 2) computing a stream plan, or an ordered sequence of stream instances from the grounded actions.

1) Grounding geometric actions with stream objects: The **`:inputs`** are determined by using the **`:geom-precondition`** field to find matching stream objects in a geometric state at a specific step in the plan. For example, a precondition for `Place(apple, rack)` is `(in-grasp apple ?g)`, where `?g` is an undetermined stream object defined in the **`:inputs`** field. The geometric state is a set of geometric propositions. If the geometric state at the beginning of `Place(apple, rack)` is {`(at-pose orange p1)`, `(in-grasp apple g1)`}, then from the precondition `(in-grasp apple ?g)`, we can infer that `g1` is a valid argument for the input parameter `?g`. The **`:outputs`** are generated by the actions, so the stream planner will simply define new stream objects for each action call. For example, the stream planner may define a stream object `p1` as the output of `Place(apple, rack)`. The action call `Place(apple, rack)` is now grounded with concrete stream objects `g1` and `p1` (these stream objects will not be assigned values until the stream sampling stage). We will represent this grounded action call as `Place(apple, rack; g1, p1)`.

2) Computing a stream plan from grounded actions: Another geometric precondition of `Place(apple, rack)` is the certified fact `(sample-pose apple rack ?p)`. When the geometric action is fully grounded with stream objects (e.g. `Place(apple, rack; g1, p1)`), then the certified facts in its preconditions can be mapped to stream instances. For example, the `sample-pose` precondition becomes `(sample-pose apple rack p1)`, which corresponds to the stream instance `sample-pose(apple, rack)`→`p1`. This precondition indicates that the successful sampling of the stream instance `sample-pose(apple, rack)`→`p1` is required for the execution of the geometric action `Place(apple, rack; g1, p1)`, and thus this stream instance is added to the stream plan ψ.

Note that during the stream planning phase, the planned stream instances are not actually sampled. The evaluation of stream instances are deferred to the stream sampling phase (e.g. PDDLStream's `Adaptive` algorithm). During stream sampling, if the stream instance `sample-pose(apple, rack)`→`p1` produces a successful sample, then the certified fact `(sample-pose apple rack p1)` gets returned, and the corresponding precondition for the geometric action `Place(apple, rack; g1, p1)` is satisfied. Otherwise, stream sampling continues until timeout, and the task planning domain will be updated with a constraint from the most recent sampling failure.

B. COAST Constraints

Our approach relies on a constraint-based feedback model where motion planning failures during plan refinement are converted to logical constraints embedded in the task planner. We observed little to no slowdown in the task-planning time from adding these constraints to the planner. We provide generalized sequence, action, and collision constraints.

1) Sequence Constraint: If the task plan [`Pick(apple, table)`, `Place(apple, rack)`] fails because `Place(apple, rack)` is infeasible, then we need the task planner to produce an alternative task plan where `Place(apple, rack)` is not attempted directly after the same preceding sequence of actions [`Pick(apple, table)`]. This is a general constraint that

requires automatically augmenting `domain.pddl` with timestamps in order to be compatible with off-the-shelf PDDL solvers. The implementation of this constraint is described in detail in the supplementary material on our webpage.

2) Action Constraint: It may be appropriate to prevent an action being executed with the same arguments, regardless of the sequence of actions preceding it. We can accomplish this by automatically augmenting every action in `domain.pddl` with the precondition that it has not failed before. Below, we show the augmented definitions of `Pick` and `Place`, where the original preconditions from the definitions in Sec. III-A are replaced with `; ...same as original` for brevity.

```
(:action Pick
  :parameters (?o - obj ?r - region)
  :precondition (and
    ; ...same as original
    (not (fail-pick ?o ?r))))
(:action Place
  :parameters (?o - obj ?r - region)
  :precondition (and
    ; ...same as original
    (not (fail-place ?o ?r))))
```

Suppose we have the task plan [`Pick(apple, table)`, `Place(apple, rack)`]. If motion planning for this task plan fails on the first action `Pick(apple, table)`, then we can prevent the task planner from proposing this action again by adding the (fail-pick apple table) proposition to the initial state s_0 in the PDDL problem. Since we maintain a queue of all previous planning states and their constraints, we will eventually revisit previous plans and maintain probabilistic completeness (Sec. VII). Note that this constraint is analogous to the failure-generalization constraint in IDTMP [4].

3) Collision Constraint: Sequence and action constraints can be applied to any domain, but for manipulation domains, we may often want a stronger constraint for handling movable obstructions. IDTMP [4] proposes a location-based constraint for the *Blocks* domain, where if picking or placing a particular block at a particular location is identified to cause a collision, then *all* blocks will be prohibited from being picked or placed at the same location. Similar to IDTMP, this constraint relies on a discrete set of pre-defined locations. We show an abbreviated action definition below for the *Block* domain.

```
(:action Pick
  :parameters (?b - block ?l - loc)
  :precondition (and
    ; ...
    (clear ?b))
  :effect (and
    ; ...
    (not (clear ?b)) (clear ?l)))
(:action Place
  :parameters (?b - block ?l - loc)
  :precondition (and
    ; ...
    (clear ?l))
  :effect (and
    ; ...
    (clear ?b) (not (clear ?l)))
)
```

We formulate this constraint as a logical implication and append the implication to the effect of the action that occurs before the failure.

Let a be the action that failed and l be the location of a collision. Our collision constraint is then the following:

$$\neg\texttt{clear}(l) \implies \texttt{fail-}a \wedge \texttt{clear}(l) \implies \neg\texttt{fail-}a$$

This means that when an action fails, we will prune out any plan that has the same collision.

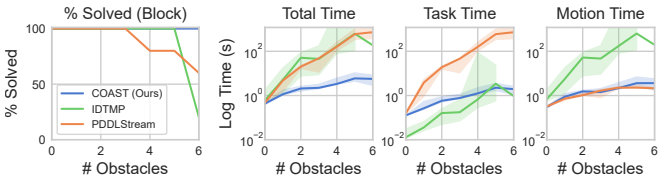

Fig. 2: Percentage solved and cumulative task and motion planning times for the *Blocks* domain with increasing number of obstacles. On the most complex configuration (6 obstacles), our algorithm achieves 100% success while IDTMP achieves 20% and PDDLStream achieves 60%. The reported planning times include the failed trials that time out at 1200s. Our algorithm solves the largest problem two orders of magnitude faster than PDDLStream and IDTMP.

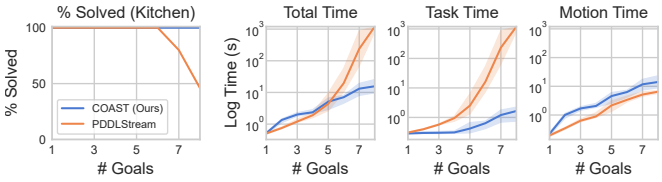

Fig. 3: Percentage solved and cumulative task and motion planning times for the *Kitchen* domain with increasing number of cook/clean goals. PDDLStream's planning process times out after 1200 seconds for 46% of the most challenging tasks (8 cook/clean goals), whereas our method achieves 100% success at magnitudes faster. PDDLStream's slow task planning times come from the explosive growth of its task state with stream objects, which our method avoids by introducing stream objects *after* task planning.

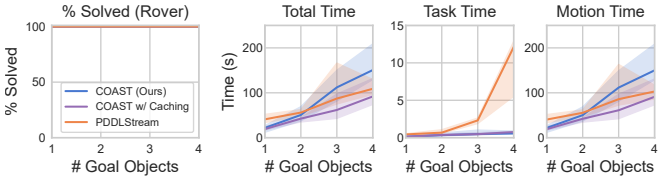

Fig. 4: Percentage solved and cumulative task and motion planning times for the *Rover* domain with increasing number of goal objects. We also include an ablation (purple) of our method with stream instance caching. PDDLStream has exponential growth in task planning time, whereas our task planning time remains nearly constant.

V. Experiments

a) Experimental Domains, Metrics and Baselines: We evaluate our approach in the three domains (Blocks [4], Kitchen [3] and Rover [3]) visualized in Fig. 1. For an in-depth explanation of these domains, we refer the reader to the supplementary material on the webpage. We compare the cumulative median task and motion planning times of our algorithm with a 50% confidence interval to that of PDDLStream's `Adaptive` Algorithm and IDTMP. For the *Blocks* domain, we compare PDDLStream, IDTMP, and our method on problems with 0–6 obstructing blocks, with 5 seeded trials for each. We run IDTMP and our method with collision constraints. We further compare our method to PDDLStream in the *Kitchen* and *Rover* domains. For *Kitchen*, we increase the difficulty by incrementing the

number of cook and clean goals from 1 to 8, each involving a variable number of the 4 items. We use our general timestep constraint and 50 seeded trials for each number of goals. For *Rover*, we scale the number of objectives and rocks (goal objects) N from 1 to 4, using our action constraint with 10 seeded trials. We institute a total planning timeout of 1200s on all domains and trials and report the percentage solved on each trial. Since *Kitchen* and *Rover* have large state spaces, we only compare them against PDDLStream because IDTMP requires a discretization of the object state space, which is not a scalable approach for these two domains.

b) Results: The results for *Blocks* are shown in Fig. 2. Overall, IDTMP's cumulative task planning time with the CSP solver is comparable in magnitude to our PDDL planning with some differences attributed to FastDownward [24] IO overhead. This demonstrates that our PDDL constraint framework is comparable in performance to IDTMP's CSP constraint framework.

PDDLStream, which also uses Fast Downward, significantly slows down with the number of blocks. The *Blocks* domain has many infeasible actions due to obstructing blocks, which is difficult for PDDLStream to handle. PDDLStream requires many stream objects and iterations of planning to support long and geometrically feasible actions. For the five obstacle case, PDDLStream and IDTMP have a median time of 600s, whereas our method takes 6s to plan. For IDTMP, motion planning is the bottleneck. To find the goal configuration for an action, IDTMP samples various collision-free goal configurations around a target object or location until a timeout or collision-free configuration is found. In contrast, in PDDLStream and our method, we calculate the inverse kinematics solution for the grasp and approach pose and perform collision-checking for the trajectory between them. This streamlines the motion planning process significantly. We express this motion planning grounding with actions, streams, and stream objects in PDDLStream's `stream.pddl` and our `geometric.pddl`.

The results for *Kitchen* are shown in Fig. 3. The *Kitchen* task involves repetitive transfer of objects between surfaces, which requires long, chained stream plans where stream instance outputs are frequently inputs to future stream instances. For PDDLStream, This requires many iterations of stream generation to produce high-level stream instances. This slows down task planning because many symbolic stream objects are added to the task state. When there are 8 goals, PDDLStream takes a total planning time of 1000s, whereas our method takes 10s. For the *Rover* domain in Fig. 4, PDDLStream spends the least amount of time on task planning compared to *Block* and *Kitchen*. This is specifically because in this domain PDDLStream's incremental stream generation algorithm can recycle rover positions across actions and iterations of planning. This requires fewer iterations of stream generation and therefore fewer stream objects in the task state, resulting in faster planning. Since we directly ground an action plan into a stream plan, every stream object will be unique, which prevents the recycling of stream objects. We circumvent this issue by introducing a caching extension to streams. As shown in the ablation in Fig 4, without caching, our method is less efficient than PDDLStream during motion planning because it cannot recycle stream results across a plan. However, with caching stream instances that have PDDL objects as inputs, we achieve similar performance. With four goal objects, our task planning time with caching is 1s compared to PDDLStream's 10s; however, our total time is only marginally better since motion planning time dominates in this domain. Overall, we show superior performance compared to PDDLStream and IDTMP on total planning time for *Blocks* and *Kitchen* and superior task planning time to PDDLStream on *Rover*.

c) Discussion: Our method requires a new formulation of writing how streams and PDDL actions are combined, but we believe this formulation is more straightforward and as expressive as PDDLStream from the results on three different domains. With this new formulation, we can remove the optimistic stream generation process and directly map action plans to stream plans. This comes at a cost of not being able to rely on task planning to recycle stream outputs across different actions of a task plan, making refinement more inefficient per plan. However, this motion planning inefficiency is insignificant compared to the performance boost gained by our task planning approach. A limitation of constraints are that they require manual engineering for each task; however, this can be a benefit since it gives a way for the user to directly embed domain knowledge to reduce the search space of TAMP.

VI. Conclusion

We present COAST, a sampling-based TAMP algorithm that significantly outperforms previous state-of-the-art algorithms in terms of task planning time for a variety of domains. The key to faster planning is our novel stream planning subroutine, which bridges vanilla PDDL constraint-based task planning with stream-based motion planning and allows us to benefit from both.

VII. Probabilistic Completeness

Theorem 7.1: For feasible problems, as the number of samples approach infinity, the probability of success of COAST will approach 1.

Proof: Given a TAMP task formulated as described in Sec. IV, where the given task planner is complete and streams are probabilistically-complete, let a feasible task plan be π_f and a feasible refinement for π_f be Y_f. Our algorithm queries the task planner and then calls the semi-complete `Adaptive` sampling algorithm [3]. `Adaptive` will eventually find a feasible refinement if it exists. If no refinement is found, we backtrack to a previous state, and attempt refinement again, continuing to attempt all unsuccessfully refined task plans. Since task planning is complete, we are guaranteed to produce π_f and since we eventually reattempt all unsuccessful refinements, our algorithm will eventually find the feasible refinement Y_f to π_f and our algorithm is probabilistically-complete. ■

REFERENCES

[1] 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.

[2] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "Sampling-based methods for factored task and motion planning," *Int. J. Rob. Res.*, vol. 37, no. 13–14, p. 1796–1825, dec 2018. [Online]. Available: https://doi.org/10.1177/0278364918802962

[3] C. R. Garrett, T. Lozano-Perez, and L. P. Kaelbling, "Pddlstream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning," in *International Conference on Automated Planning and Scheduling*, 2018.

[4] 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*, 2018.

[5] W. Thomason, M. P. Strub, and J. D. Gammell, "Task and motion informed trees (tmit*): Almost-surely asymptotically optimal integrated task and motion planning," *IEEE Robotics and Automation Letters*, vol. 7, no. 4, pp. 11 370–11 377, 2022.

[6] R. Chitnis, T. Silver, J. B. Tenenbaum, T. Lozano-Pérez, and L. P. Kaelbling, "Learning neuro-symbolic relational transition models for bilevel planning," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2022, pp. 4166–4173.

[7] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, *PDDL—The Planning Domain Definition Language Version 1.2*, 1998.

[8] I. Georgievski and M. Aiello, "Htn planning: Overview, comparison, and beyond," *Artificial Intelligence*, vol. 222, pp. 124–156, 2015.

[9] R. Lallement, L. de Silva, and R. Alami, "Hatp: An htn planner for robotics," *2nd ICAPS Workshop on Planning and Robotics*, 2014.

[10] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "Backward-forward search for manipulation planning," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2015.

[11] J. Wolfe, B. Marthi, and S. J. Russell, "Combined task and motion planning for mobile manipulation," in *International Conference on Automated Planning and Scheduling*, 2010.

[12] L. Karlsson, J. Bidot, F. Lagriffoul, A. Saffiotti, U. Hillenbrand, and F. Schmidt, "Combining task and path planning for a humanoid two-arm robotic system," in *ICAPS Workshop on Combining Task and Motion Planning for Real-World Applications*, 2012.

[13] 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.

[14] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," in *International Joint Conference on Artificial Intelligence*, 1971.

[15] S. Liu and P. Liu, "A review of motion planning algorithms for robotic arm systems," November 2020.

[16] 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*, 2015.

[17] T. Migimatsu and J. Bohg, "Object-centric task and motion planning in dynamic environments," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 844–851, 2020.

[18] D. Driess, J.-S. Ha, and M. Toussaint, "Learning to solve sequential physical reasoning problems from a scene image," *The International Journal of Robotics Research*, 2021.

[19] 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*, 2021.

[20] 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*, 2021.

[21] R. Chitnis, D. Hadfield-Menell, A. Gupta, S. Srivastava, E. Groshev, C. Lin, and P. Abbeel, "Guided search for task and motion plans using learned heuristics," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2016, pp. 447–454.

[22] A. Curtis, X. Fang, L. P. Kaelbling, T. Lozano-Pérez, and C. R. Garrett, "Long-horizon manipulation of unknown objects via task and motion planning with estimated affordances," in *2022 International Conference on Robotics and Automation (ICRA)*, 2022, pp. 1940–1946.

[23] 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)*. IEEE, 2020, pp. 5678–5684.

[24] M. Helmert, "The fast downward planning system," *Journal of Artificial Intelligence Research*, vol. 26, pp. 191–246, 2006.