Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)

PDDLStream: Integrating Symbolic Planners and Blackbox Samplers via Optimistic Adaptive Planning

Caelan Reed Garrett, Tomás Lozano-Pérez, Leslie Pack Kaelbling
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{caelan, tlp, lpk}@csail.mit.edu*

Abstract

Many planning applications involve complex relationships defined on high-dimensional, continuous variables. For example, robotic manipulation requires planning with kinematic, collision, visibility, and motion constraints involving robot configurations, object poses, and robot trajectories. These constraints typically require specialized procedures to sample satisfying values. We extend PDDL to support a generic, declarative specification for these procedures that treats their implementation as black boxes. We provide domain-independent algorithms that reduce PDDLStream problems to a sequence of finite PDDL problems. We also introduce an algorithm that dynamically balances exploring new candidate plans and exploiting existing ones. This enables the algorithm to greedily search the space of parameter bindings to more quickly solve tightly-constrained problems as well as locally optimize to produce low-cost solutions. We evaluate our algorithms on three simulated robotic planning domains as well as several real-world robotic tasks.

1 Introduction

Many important planning domains occur in continuous spaces involving complex constraints among variables. Consider planning for an 11 degree-of-freedom (DOF) robot tasked with rearranging blocks. The robot must find a sequence of `move`, `pick`, and `place` actions involving continuous variables such as robot configurations, robot trajectories, block poses, and block grasps that satisfy complicated kinematic, collision, visibility, and motion constraints, which affect the feasibility of the actions. Often, special purpose procedures for evaluating and producing satisfying values for these constraints, such as inverse kinematic solvers, collision checkers, and motion planners, are known.

We propose PDDLStream, a planning language that introduces *streams* as an interface for incorporating sampling procedures in Planning Domain Definition Language (PDDL) (McDermott et al. 1998). Streams have both a procedural and declarative component. The procedural component is a *conditional generator*, a function from input values to a possibly infinite sequence of output values. Conditional generators construct new values that depend on existing values, such as new robot configurations that satisfy a kinematic constraint with existing poses and grasps. The declarative component specifies the facts that these input and output values satisfy. Streams allow a planner to reason about conditions on the inputs and outputs of a conditional generator while treating its implementation as a black box.

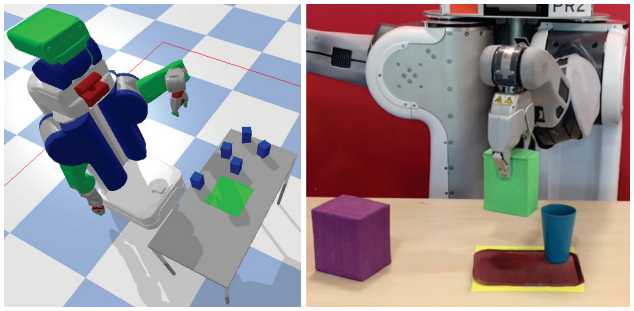

Figure 1: Left: *Domain 1* (with 5 blocks). Right: A real-world robot planning to "serve a meal" on the brown tray.

We apply two *existing* algorithms (Garrett, Lozano-Pérez, and Kaelbling 2018) to PDDLStream and introduce two *new* PDDLStream algorithms. Each algorithm constructs and solves a sequence of finite PDDL problems, *any* off-the-shelf PDDL planner to be used as a search subroutine. Our *Adaptive* algorithm balances the *exploration-exploitation trade-off* (Robbins 1952) when deciding whether to search for new *optimistic* plans or to continue sampling parameter values for existing ones. By adaptively balancing the time spent searching versus sampling, *Adaptive* is often able to more aggressively find parameter *bindings* for existing optimistic plans. We experiment in three robotic planning domains (figures 1, 2, and 3) to compare the algorithms. *Adaptive* greatly outperforms the two existing algorithms (Garrett, Lozano-Pérez, and Kaelbling 2018) on constrained and

*We gratefully acknowledge support from NSF grants 1523767 and 1723381; from AFOSR grant FA9550-17-1-0165; from ONR grant N00014-18-1-2847; from the Honda Research Institute; and from SUTD Temasek Laboratories. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

cost-sensitive problems. Finally, we apply PDDLStream to a real-world robot to plan for manipulation and kitchen tasks.

2 Related Work

Several PDDL extensions such as PDDL2.1 (Fox and Long 2003) and PDDL+ (Fox and Long 2006) support planning with numeric variables that evolve over time. Most numeric planners are limited to problems with linear or polynomial dynamics (Hoffmann 2003; Bryce et al. 2015; Cashmore et al. 2016); however, some planners can handle non-polynomial dynamics by discretizing time (Della Penna et al. 2009; Piotrowski et al. 2016). While it may be technically possible to analytically model, for example, collision constraints among 3D meshes using PDDL+, the resulting encoding would be enormous, far exceeding the capabilities of numeric planners. One approach addresses problems with convex dynamics without discretization (Fernández-González, Williams, and Karpas 2018); however, it requires a convex decomposition of the robot's configuration space, which is intractable for 3D articulated robots.

Semantic attachments (Dornhege et al. 2009b; 2009a; Gregory et al. 2012; Hertle et al. 2012; Dornhege 2014; Dantam et al. 2016), functions computed by an external module, are an existing method that integrates blackbox procedures and PDDL planners. Condition-checker modules test Boolean action preconditions, and effect-applicator modules modify numeric state variables. Actions must be parameterized by finite types, which restricts the technique to finite action spaces. In the context of robotics, this restricts the applicability of semantic attachments to domains that are *prediscretized*, where a *human* specifies a finite set of object poses, object grasps, and robot configurations that can be considered. Thus, semantic attachments are not sufficient for modeling the domains we consider, where the *planner* must produce these continuous values. In contrast, PDDLStream is able to model domains with infinitely-many action instances. Finally, semantic attachments are evaluated *eagerly* (section 6) during the forward state-space search as opposed to *lazily* (section 7). This results in many unneeded module calls and thus poor planner performance when the attachments are computationally expensive.

Many approaches to robotic task and motion planning have developed strategies for handling continuous spaces that go beyond prediscretization (Kaelbling and Lozano-Pérez 2011; Srivastava et al. 2014; Garrett, Lozano-Pérez, and Kaelbling 2015; Toussaint 2015; Garrett, Lozano-Pérez, and Kaelbling 2017). However, these approaches are each specialized to a particular class of manipulation problems. Moreover, they cannot be applied new domains, such as the rovers domain in figure 2, without substantial engineering effort because they do not offer a modular, domain-agnostic problem description language with clear semantics.

3 PDDLStream

We build PDDLStream on PDDL (McDermott et al. 1998) to enable ease of use for AI practitioners as well as to leverage any PDDL planner, without modification, as a subroutine. We provide an example PDDLStream specification for a robotic pick-and-place domain in section 4. For clarity of exposition, we formalize STRIPS (Fikes and Nilsson 1971) PDDL problems; however, our approach also applies to Action Description Language (ADL) (Pednault 1989) features such as typing, disjunctions, negative preconditions, existential quantifiers, finite universal quantifiers, conditional effects, and derived predicates.

A *predicate* p is a Boolean function. We treat *types* as unary predicates. An atomic *fact* $p(\bar{x})$ is a predicate p evaluated on *object* tuple $\bar{x} = \langle x_1, ..., x_k \rangle$ that evaluates to true. A *literal* is a fact or a negated fact. A *state* $\mathcal{I}$ is a set of literals. By the closed world assumption, facts not explicitly specified within a state are false. An *action* a is given by a *parameter* tuple $\bar{X} = \langle X_1, ..., X_k \rangle$, a set of literal *preconditions* $pre(a)$ on $\bar{X}$, and a set of literal *effects* $\mathit{eff}(a)$ on $\bar{X}$. In *cost-sensitive* planning, each action may have a nonnegative *cost function* $c(\bar{X})$ as an additive cost term. An *action instance* $a(\bar{x})$ is an action a with its parameters $\bar{X}$ replaced with objects $\bar{x}$. An action instance $a(\bar{x})$ is *applicable* in a state $\mathcal{I}$ if $(\mathit{pre}^+(a(\bar{x})) \subseteq \mathcal{I}) \wedge (\mathit{pre}^-(a(\bar{x})) \cap \mathcal{I} = \emptyset)$ where the $+$ and $-$ superscripts designate the positive and negative literals respectively. The result of *applying* an action instance $a(\bar{x})$ to state $\mathcal{I}$ is a new state $(\mathcal{I} \setminus \mathit{eff}^-(a(\bar{x}))) \cup \mathit{eff}^+(a(\bar{x}))$. To compactly model the domain in section 4, we make use of *derived predicates* (*axioms*) (Fox and Long 2003; Thiébaux, Hoffmann, and Nebel 2005), which are defined by a logical formula on a state. We treat positive-mentioned instantiated axioms roughly as actions for the purpose of describing the algorithms. A STRIPS PDDL *problem* $(\mathcal{A}, \mathcal{I}, \mathcal{G})$ is given by a set of actions $\mathcal{A}$, an initial state $\mathcal{I}$, and a goal set of literals $\mathcal{G}$. A *plan* $\pi = [a_1(\bar{x}_1), ..., a_k(\bar{x}_k)]$ is a finite sequence of k action instances such that each $a_i(\bar{x}_i)$ is applicable in the $(i-1)$th state resulting from their application. The *preimage* of a consistent plan π is the set of facts that must hold to make π executable:

$$\text{PREIMAGE}(\pi) = \bigcup_{i=1}^{k} \Big(\mathit{pre}(a_i(\bar{x}_i)) - \bigcup_{j<i} \mathit{eff}(a_j(\bar{x}_j)) \Big).$$

3.1 Streams

A *generator* $g = [\bar{y}_1, \bar{y}_2, ...]$ is a finite or infinite, enumerable sequence of object tuples $\bar{y}_i$. Let $\textbf{next}(g)$ *evaluate* the generator and return the subsequent $\bar{y}_i$ in the sequence if it exists. Otherwise, let $\textbf{next}(g)$ return **None**. Let $\textbf{count}(g) = i$ return the current number of times $\textbf{next}(g)$ has been called. A *conditional generator* $f(\bar{X})$ is a function from an object tuple $\bar{x}$ to a generator $f(\bar{x}) = g_{\bar{x}}$ that produces a sequence of *output* object tuples $g_{\bar{x}}$ that relate to *input* object tuple $\bar{x}$.

A *stream* s is a conditional generator $s(\bar{X})$ endowed with a declarative specification of any facts its inputs and outputs always satisfy. Let $s.\mathit{domain} = \{p \mid \forall \bar{x} \in \bar{X}.\ p(\bar{x})\}$ be a set of facts p on input parameters $s.\mathit{input}$ that specify the set of object tuples $\bar{x}$ for which $s(\bar{X})$ is defined. Let $s.\mathit{certified} = \{p \mid \forall \bar{x} \in \bar{X}, \forall \bar{y} \in s(\bar{x}).\ p(\bar{x} + \bar{y})\}$ be a set of *certified* predicates on both $s.\mathit{input}$ and output parameters $s.\mathit{output}$ that assert any facts that $\langle \bar{x}, \bar{y} \rangle$ pairs satisfy. Intuitively, domain facts specify "typing" information by declaring legal inputs, and certified facts declare properties that all outputs are guaranteed to satisfy. A *stream instance* $s(\bar{x})$ is

a stream s with its input parameters $s.input$ replaced by an object tuple $\bar{x}$. Let $s(\bar{x}) \rightarrow \bar{y}$ denote a stream instance $s(\bar{x})$ that generates output object tuple $\bar{y}$. An *external cost function* $c(\bar{X}) \rightarrow [0, \infty)$ is a nonnegative function defined on parameter tuple $\bar{X}$. Like streams, the domain of c is declared by a set of facts $c.domain$ on inputs $\bar{X}$. However, external cost functions do not produce objects or certify facts.

A PDDLStream *problem* $(\mathcal{A}, \mathcal{S}, \mathcal{I}, \mathcal{G})$ is given by a set of actions $\mathcal{A}$, a set of streams $\mathcal{S}$, an initial state $\mathcal{I}$, and a goal state set $\mathcal{G}$. To ensure PDDLStream is Turing-recognizable, we require that stream-certified predicates are never negated within action preconditions. The set of streams $\mathcal{S}$ augments the initial state $\mathcal{I}$, recursively defining a potentially infinite set of facts $\mathcal{I}^*$ that hold initially and cannot be changed:

$$\mathcal{I}^* = \mathcal{I} \cup \{p(\bar{x} + \bar{y}) \mid s \in \mathcal{S}, |\bar{x}| = |s.input|,$$
$$\forall p' \in s.domain.\ p'(\bar{x}) \in \mathcal{I}^*, \bar{y} \in s(\bar{x}), p \in s.certified\}.$$

A *solution* π for PDDLStream problem $(\mathcal{A}, \mathcal{S}, \mathcal{I}, \mathcal{G})$ is a plan such that $\text{PREIMAGE}(\pi + [\mathcal{G}]) \subseteq \mathcal{I}^*$. For cost-sensitive planning, the objective is to minimize the sum of solution action costs. In the extended version of this paper (https://arxiv.org/abs/1802.08705), we prove that PDDLStream planning is *undecidable*, but prove our algorithms are *semi-complete*, *i.e.*, complete over feasible instances.

3.2 Domain Description

In order to enable easy use for AI practitioners, PDDLStream adheres to the PDDL standard when possible and adapts PDDL style and syntax when describing streams. PDDL problems are typically described using text files. A `domain.pddl` file specifies the domain dynamics through a set of actions (`:action`) and derived predicates (`:derived`). A `problem.pddl` file specifies the problem instance through a set of objects (`:objects`), the initial state (`:init`), and a goal formula (`:goal`).

In order to represent first-class objects such as real-valued vectors and implement conditional generators that operate on them, PDDLStream problems are partially described using a programming language. However, the declarative components of PDDLStream are still described in PDDL. Actions and derived predicates are listed using a standard `domain.pddl` text file. The input parameters (`:inp`), domain facts (`:dom`), output parameters (`:out`), and certified facts (`:cert`) of each stream are specified in a `stream.pddl` text file using PDDL-style syntax.

The conditional generator for each stream is stored programmatically in a map from each stream name to its generator function. Because the initial state typically contains many constant objects that may be non-string entities, the initial state and goal formula are also expressed programmatically instead of using a `problem.pddl` text file.

4 Example Domains

We apply PDDLStream to model two robotic manipulation domains with a single manipulator and a finite set of movable blocks. *Domain 1* (figure 1) is mobile manipulation task requiring a PR2 robot to tightly pack each blue block into the green region. The goal in *Domain 2* (figure 3) is to place one of the two blue blocks on the green region while minimizing the robot distance traveled. The right blue block is much closer to the robot and the goal region than the distant left blue block. However, the red block must be moved out of way in order to safely grasp the right blue block. Optimal plans, which pick the near blue block, require more actions but travel less distance than plans that pick the far blue block.

Our model uses the following parameters: `?b` is the name of a block; `?r` is the name of a region on a stable surface; `?p` is 6 DOF block pose placed stably on a fixed surface; `?g` is a 6 DOF block grasp transform relative to the robot gripper; `?q` is an 11 DOF robot configuration; and `?t` is a trajectory composed of a finite sequence of waypoint robot configurations. The fluent predicates `AtConf`, `AtPose`, `Holding`, `Empty` model the changing robot configuration, object poses, and gripper status. The static predicates `Block`, `Conf`, `Pose`, `Grasp`, `Kin`, `Motion`, `Contain`, `CFree` are constant facts. `Block` declares that `?b` is a block. `Conf` declares that `?q` is a robot configuration. `Pose` and `Grasp` indicate that a pose `?p` or grasp `?g` can be used for block `?b`. `Kin` is a kinematic constraint. `Motion` is a constraint that `?q1`, `?q2` are the start and end configurations for trajectory `?t`, and `?t` respects joint limits, self-collisions, and collisions with the fixed environment. `Contain` states that when block `?b` is at pose `?p`, it is within region `?r`. `CFree` states that if block `?b` were placed at pose `?p`, the robot, executing trajectory `?t`, would not collide with it. The cost function `Dist` gives the distance traveled along trajectory `?t`. The `domain.pddl` file is specified as follows:

```
(:derived (In ?b ?r)
 (exists (?p) (and (Contain ?b ?p ?r)
                   (AtPose ?b ?p))))
(:derived (Safe ?t ?b) (or
 (exists (?g) (and (Grasp ?b ?g)
                   (Holding ?b ?g)))
 (exists (?p) (and (CFree ?t ?b ?p)
                   (AtPose ?b ?p)))))
(:action move
 :param (?q1 ?t ?q2)
 :pre (and (Motion ?q1 ?t ?q2) (AtConf ?q1)
(forall(?b)(imply (Block ?b) (Safe ?t ?b))))
 :eff (and (AtConf ?q2) (not (AtConf ?q1))
      (incr (total-cost) (Dist ?t)))
(:action pick
 :param (?b ?p ?g ?q)
 :pre (and (Kin ?b ?p ?g ?q) (AtPose ?b ?p)
           (Empty) (AtConf ?q))
 :eff (and (Holding ?b ?g)
      (not (AtPose ?b ?p)) (not (Empty))))
(:action place
 :param (?b ?p ?g ?q)
 :pre (and (Kin ?b ?p ?g ?q) (Holding ?b ?g)
           (AtConf ?q))
 :eff (and (AtPose ?b ?p) (Empty)
      (not (Holding ?b ?g))))
```

Three actions are defined: `move`, `pick`, and `place`. The `In` derived predicate expresses whether block `?b` is currently contained within region `?r` by expressing a condition on its current pose `?p`. The `Safe` derived predicate encodes whether trajectory `?t` does not collide with placed block `?b`

at its current pose. For simplicity, we omit the description of an additional condition within `move` that checks collisions between grasped blocks and placed blocks.

The `stream.pddl` file is defined below. The `poses` stream randomly samples an infinite sequence of stable placements `?p` for block `?b` in region `?r`. The `grasps` stream enumerates a sequence of force-closure grasps `?g` for block `?b`. The `ik` stream calls an inverse kinematics solver to sample configurations `?q` from a 4D manifold of values (due to manipulator redundancy) that enable the robot to manipulate a block `?b` at pose `?p` with grasp `?g`. It is important for `ik` to have `?p` and `?g` as input parameters so it can operate on poses and grasp objects in the initial state as well those produced by `poses` and `grasps`. The `motion` stream repeatedly calls a motion planner to generate safe trajectories `?t` between pairs of configurations `?q1`, `?q2`. The `cfree` stream tests whether block `?b` when at pose `?p` is collision free with respect to all robot configurations along trajectory `?t`. It is a *test stream*, a stream with no output parameters. If it generates the empty tuple $\langle\ \rangle$, its certified conditions are proven. As a result, it can be interpreted as a Boolean function. `cfree` is checked by calling a collision checker along trajectory `?t`. The `Dist` external cost function returns the sum of the distance between each pair of adjacent configuration waypoints on trajectory `?t`.

```
(:stream poses
 :inp (?b ?r)
 :dom (and (Block ?b)
 (Region ?r))
 :out (?p)
 :cert (and (Pose ?b ?p)
 (Contain ?b ?p ?r)))
(:stream grasps
 :inp (?b)
 :dom (Block ?b)
 :out (?g)
 :cert (Grasp ?b ?g))
(:stream cfree
 :inp (?t ?b ?p)
 :dom (and (Traj ?t)
 (Pose ?b ?p))
 :cert (CFree ?t ?b ?p))
```

```
(:stream ik
 :inp (?b ?p ?g)
 :dom (and
 (Pose ?b ?p)
 (Grasp ?b ?g))
 :out (?q)
 :cert (and (Conf ?q)
 (Kin ?b ?p ?g ?q)))
(:stream motion
 :inp (?q1 ?q2)
 :dom (and (Conf ?q1)
 (Conf ?q2))
 :out (?t)
 :cert (and (Traj ?t)
 (Motion ?q1 ?t ?q2)))
(:function (Dist ?t)
 :dom (Traj ?t))
```

4.1 Rovers Domain

We also apply PDDLStream to a multi-robot surveying domain to demonstrate the generality of our formalism. *Domain 3* (figure 2) extends the classic PDDL domain *rovers* (Long and Fox 2003) by incorporating 3D visibility, distance, reachability, and collision constraints. Two rovers (green TurtleBot robots) must together collect a rock sample (black objects), collect a soil sample (brown objects), photograph each objective (blue objects) without occlusions, and communicate the results back to the lander (yellow Husky robot) via line of sight. Due to obstacles that limit reachability, both rovers must be utilized in order to complete the task. The actions are: `move`, `take_image`, `calibrate`, `send_image`, `sample_rock`, `send_analysis`, `drop_rock`.

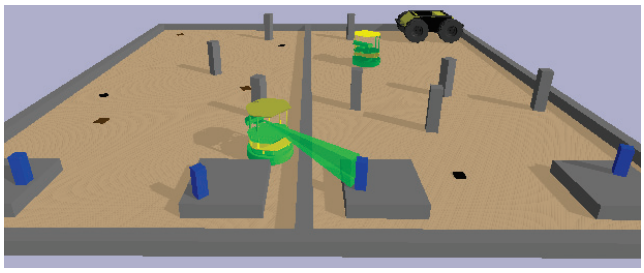

Figure 2: *Domain 3* (with 4 objectives).

5 PDDLStream Algorithms

We present four PDDLStream algorithms that share several common subroutines. The first two algorithms (*Incremental*, *Focused*) are the direct application of the algorithms of Garrett et al. (2018) to PDDLStream. The second two algorithms (*Binding*, *Adaptive*) are new algorithms. Each algorithm operates by solving a sequence of finite PDDL problems of increasing size. Let $\text{SEARCH}(\mathcal{A}, \mathcal{I}, \mathcal{G})$ be any sound and complete algorithm for classic PDDL problems. For cost-sensitive planning, assume SEARCH returns a solution with cost below a cost bound C. SEARCH can be implemented using an off-the-shelf PDDL planner without modification to take advantage of existing, efficient search algorithms. Although each algorithm is presented in its decision form, each can easily be run in an *anytime* fashion.

In order to reduce a potentially infinitely-large PDDLStream problem to a sequence of finite PDDL problems, our algorithms control two infinite sources of objects. First, the generator for a stream instance may enumerate an infinitely large set. Second, it may be possible to compose a sequence of stream instances of unbounded length. Thus, both the maximum *width* and *depth* of generated objects must be limited. We capture both of these properties by introducing the notion of the *level* of a fact. Intuitively, a level relates to the number of stream evaluations that are required to certify a fact. The level recursively incorporates both the stream evaluations required to certify its domain facts as well as the number of evaluations of $s(\bar{x})$ itself. This idea is similar to the concept of layer for facts and actions in a relaxed planning graph (Bonet and Geffner 2001) with the distinction that a stream instance can be evaluated many times.

Each algorithm maintains a map U from each certified fact to both the level ($level$) of the fact and the stream instance ($instance$) that certified it. More formally, the level of stream instance $s(\bar{x})$ is the maximum level of its domain facts in U plus one more than the count of its past evaluations. See section 8.2 for an example using levels.

$$\text{LEVEL}(U, s(\bar{x})) = 1 + \textbf{count}(s(\bar{x})) + \max_{p \in s.domain} U[p(\bar{x})].level$$

To ensure that external cost functions are evaluated on the earliest level possible, define the level of an external cost function instance $c(\bar{x})$ to be the max of its domain, *i.e.* $\text{LEVEL}(U, c(\bar{x})) = \max_{p \in f.domain} U[p(\bar{x})].level$.

6 Incremental Algorithm

The *Incremental* algorithm enumerates $\mathcal{I}^*$ by iteratively increasing the maximum level l. For each level, the subroutine APPLY-STREAMS instantiates and evaluates all stream instances $s(\bar{x})$ at level $k \leq l$ and adds any new certified facts to U. The meta-parameter OUTPUT specifies the procedure that is used to generate output objects when evaluating each stream instances. In this case, OUTPUT = **next** simply queries the next output tuple in the generator. Let procedure INSTANTIATE ground all stream instances that are legal given the input objects in U and the currently certified facts:

$$\text{INSTANTIATE}(\mathcal{S}, U) = \{s(\bar{x}) \mid \forall s \in \mathcal{S}, \forall p \in s.domain. \\ |\bar{x}| = |s.input|, p(\bar{x}) \in U\}.$$

The current set of certified facts U becomes the initial state in a PDDL problem $(\mathcal{A}, U, \mathcal{G})$ that is solved using SEARCH. If SEARCH finds a plan π, it is returned as a solution.

```
INCREMENTAL(A, S, I, G) :
1  U = {f : ⟨0, None⟩; f ∈ I} // Map from fact to level
2  for l ∈ [0, 1, 2, ...]:
3      U = APPLY-STREAMS(S, U, l; next)
4      π = SEARCH(A, U, G)
5      if π ≠ None: return π

APPLY-STREAMS(S, U', l; OUTPUT) :
1  U = copy(U')
2  for k ∈ [1, 2, ..., l]:
3      for s(x̄) ∈ INSTANTIATE(S, U): if LEVEL(U, s(x̄)) = k
4          ADD-CERTIFIED(U, s(x̄); OUTPUT)
5  return U

ADD-CERTIFIED(U, s(x); OUTPUT) :
1  l = LEVEL(U, s(x̄)); ȳ = OUTPUT(x̄)
2  F = {p(x̄ + ȳ) | p ∈ s.certified} if ȳ ≠ None else ∅
3  for f ∈ (F \ U): U[f] = ⟨l, s(x̄)⟩
4  return ȳ
```

The incremental algorithm *eagerly* and blindly evaluates all stream instances, producing many facts that are irrelevant to the task. This can result in significant overhead when stream evaluations are computationally expensive as they frequently are in robotics domains where inverse kinematics solvers and motion planners are required.

7 Optimistic Algorithms

The remaining algorithms (*Focused*, *Binding*, and *Adaptive*) use the shared pseudocode OPTIMISTIC, which takes in a meta-parameter procedure PROCESS-STREAMS that implements each algorithm. The key principle behind our algorithms is to *lazily* explore candidate plans before checking their validity (Dellin and Srinivasa 2016). In order to apply laziness to PDDLStream, we plan using *optimistic objects* that represent hypothetical stream outputs before evaluating actual stream outputs. These values are optimistic in the sense that their corresponding stream instance may not ever produce a satisfying value. For instance, an `ik` stream with a particular pose and grasp pair as inputs may not admit any inverse kinematic solutions. By first planning with optimistic objects, our algorithms are able to identify only the stream instances that could possibly support a plan and therefore focus sampling on useful aspects of the problem.

Let the procedure OPT-OUTPUT$(s(\bar{x})) = \bar{o}^s_x$ create an optimistic object tuple for stream instance $s(\bar{x})$. Critically, this technique differs from the approach of Garrett et al. (2018) in that here each optimistic object $\bar{o}^s_x$ is *unique* to a single stream instance $s(\bar{x})$. In contrast, the approach of Garrett et al., if directly applied to PDDLStream, would create an optimistic object tuple OPT-OUTPUT$(s(\bar{x})) = \bar{o}^s$ for each *stream* rather than each *stream instance*. As a result, $\bar{o}^s$ is *shared* among all instances of stream s. This distinction is significant because each *unique* optimistic object $\bar{o}^s_x$ implicitly encodes a single partially-ordered set of stream instance evaluations that could produce values for the optimistic object. This property provides the basis for our novel *Binding* (section 8.1) and *Adaptive* (section 8.3) algorithms.

A consequence of creating unique optimistic objects is that the set of all optimistic objects may be infinitely large in domains where it is possible to compose arbitrarily many streams instances. In contrast, creating shared optimistic objects always results in a finite set of optimistic objects. In order to limit the number of unique optimistic objects, we regulate the current set of optimistic stream instances using their level (section 5). Namely, we iteratively increase the maximum optimistic stream level l that can be considered on a given iteration. Finally, when applied to an external cost function instance $c(\bar{x})$, let OPT-OUTPUT$(c(\bar{x})) = 0$ produce an optimistic evaluation of $c(\bar{x})$ by returning 0, a lower bound on the nonnegative cost function value.

```
OPTIMISTIC(A, S, I, G; PROCESS-STREAMS) :
1  U = {f : ⟨0, None⟩ | f ∈ I} // Map from fact to level
2  for l ∈ [0, 1, 2, ...]:
3      while True:
4          U* = APPLY-STREAMS(S, U, l; OPT-OUTPUT)
5          π* = SEARCH(A, U*, G)
6          if π* = None: break
7          ψ = RETRACE(U, U*, PREIMAGE(π* + [G]))
8          π = PROCESS-STREAMS(U, ψ, π*)
9          if π ≠ None: return π

RETRACE(U, U*, F) :
1  ψ = [ ] // Initialize stream plan
2  for f ∈ (F \ U):
3      s(x̄) = U*[f].instance
4      ψ += RETRACE(U, U*, {p(x̄); p ∈ s.domain})+[s(x̄)]
5  return ψ
```

The outer loop of OPTIMISTIC iteratively increases the maximum fact level l. The inner loop identifies all stream instances at fact level l that optimistically support a plan. On each iteration of the while loop, APPLY-STREAMS instantiates and optimistically evaluates all stream instances $s(\bar{x})$ at level $k \leq l$, this time using OUTPUT = OPT-OUTPUT. This results in U^*, a map of all optimistic facts achievable at fact level l. Next, OPTIMISTIC calls SEARCH to find an optimistic plan π^* for the PDDL problem $(\mathcal{A}, U^*, \mathcal{G})$. If $\pi^* =$ **None**, no more plans can be found at the current fact level. And so OPTIMISTIC breaks out of the while loop and increases the fact level to $l + 1$. Otherwise, RETRACE extracts a *stream plan* ψ of stream instances that, presuming successful eval-

uations, certify the optimistic facts present in the preconditions of π^*. For each optimistic fact in the preimage of π^*, RETRACE adds the stream instance $s(\bar{x})$ that produced it to ψ and recursively applies RETRACE to the domain facts of $s(\bar{x})$. Once a stream plan ψ is identified, the meta-parameter procedure PROCESS-STREAMS evaluates a subsequence of ψ and returns a solution π if one is found.

7.1 Focused Algorithm

The *Focused* algorithm implements PROCESS-STREAMS using the procedure FOCUSED-PROCESS-STREAMS. If $\psi = []$, the plan π^* uses no optimistic objects and is returned as a solution. Otherwise, FOCUSED-PROCESS-STREAMS evaluates streams instances that have satisfied domain facts and adds new certified facts to U. The first stream instance $\psi[0]$ is always evaluated. Because evaluation with **next** increments the level of $s(\bar{x})$, the same stream plan ψ cannot be used on the following iteration. This forces SEARCH to find an optimistic plan π^* supported by a new stream plan or report that no more exist, causing the level l to increase.

FOCUSED-PROCESS-STREAMS(U, ψ, π^*) :
1 **if** $\psi = [\,]$: **return** π^*
2 **for** $s(\bar{x}) \in \psi$: **if** $\{p(\bar{x}) \mid p \in s.domain\} \subseteq U$:
3 ADD-CERTIFIED$(U, s(\bar{x})$; **next**$)$
4 **return None**

8 Binding and Adaptive Algorithms

The primary shortcoming of *Focused* is that it fails to take full advantage of the plans produced by SEARCH. Our two new algorithms implement PROCESS-STREAMS by operating on more of the associated stream plans at a time. Ultimately, our *Adaptive* algorithm balances the time spent in SEARCH versus PROCESS-STREAMS, often reducing the number of calls to SEARCH required to find a solution.

8.1 Binding Algorithm

The key idea of *Binding* is to propagate stream outputs that are inputs to subsequent streams to evaluate more of the stream plan at once. PROCESS-STREAMS-BINDING maintains a set of *bindings* B, assignments of each optimistic object to an actual object, that are produced while evaluating the stream plan ψ. Bindings are used to replace any optimistic objects that serve as stream instance inputs in ψ or action arguments in π^*. Recall from section 7 that optimistic objects $o^s_{\bar{x}}$ are *unique* to a particular stream instance $s(\bar{x})$. Thus, there is a bijective mapping between each optimistic object $o^s_{\bar{x}}$ and its corresponding output object from $s(\bar{x})$. The procedure UPDATE-BINDINGS substitutes the optimistic objects in $\bar{x}^*$ with their bindings $\bar{x}$ from B, evaluates the stream instance $s(\bar{x})$, and if an output tuple $\bar{y} \neq$ **None** is produced, updates B by mapping each optimistic output y^* to its new object y. If all stream evaluations are successful, then ψ is *satisfied*, and procedure APPLY-BINDINGS substitutes each optimistic object within π^* with its value in B and returns the new plan as a solution. If a stream instead returns **None** or the evaluated cost exceeds the current cost bound C, BINDING-PROCESS-STREAMS terminates early to avoid unnecessarily evaluating any subsequent stream instances.

BINDING-PROCESS-STREAMS(U, ψ, π^*) :
1 $B = \{\,\}$ // Initialize bindings
2 **for** $s(\bar{x}^*) \in \psi$:
3 $B =$ UPDATE-BINDINGS$(B, s(\bar{x}))$
4 **if** $B =$ **None**: **return None**
5 **return** APPLY-BINDINGS(B, π^*)

UPDATE-BINDINGS$(B, s(\bar{x}^*))$:
1 $\bar{x} = [B[x^*]$ **if** $x^* \in B$ **else** x^* **for** $x^* \in \bar{x}^*]$
2 $\bar{y} =$ ADD-CERTIFIED$(U, s(\bar{x})$; **next**$)$
3 **if** $\bar{y} =$ **None**: **return None**
4 **for** $y^*, y \in$ **zip**(OPT-OUTPUT$(s(\bar{x})), \bar{y})$: $B[y^*] = y$
5 **return** B

The performance of *Binding* depends on the number times BINDING-PROCESS-STREAMS fails to bind each stream plan ψ that is considered. And the likelihood that BINDING-PROCESS-STREAMS fails depends on the properties of the streams specified for a domain, such as the fraction of stream instances that fail to produce output values (**next**$(s(\bar{x})) =$ **None**), as well as the objects present in a specific problem instance. For example, in *Domain 1*, the first optimistic plan considered is always satisfiable; however, most calls to BINDING-PROCESS-STREAMS fail due to fact that the `cfree` stream often fails due to the highly-constrained nature of packing blocks into a small region. In *Domain 2*, the first optimistic plan is never satisfiable because the red block obstructs all ways of picking the blue block, but an optimistic plan that first moves the red block and then the blue block admits many bindings. In *Domain 3*, if a rover configuration sampled to photograph a particular objective is not reachable, it is likely that most configurations sampled for that particular rover and objective pair are not reachable.

8.2 Example Execution

As an example of BINDING-PROCESS-STREAMS, consider a PDDLStream problem in the robotics domain (section 4) requiring that block `b` be moved from initial pose p_0 to a goal region `r`. The objects $q_0, p_0, g_1, t_1, ...$ are real-valued vectors (*e.g.* $q_0 = [1.71, -2.44, ...]$). The initial state is:

$\mathcal{I} = \{$`(Region r) (Block b) (Pose b` p_0`) (Conf` q_0`) (AtPose b` p_0`) (Empty) (AtConf` q_0`)`$\}$.

The goal is $\mathcal{G} = \{$`(InRegion b r)`$\}$. OPTIMISTIC fails to find a plan for level $l \leq 2$. When $l = 3$, the optimistic stream instances instantiated by APPLY-STREAMS are:

$[\texttt{grasps(b)}\rightarrow\boldsymbol{\gamma}_1, \texttt{poses(b, r)}\rightarrow\boldsymbol{\rho}_1, \texttt{ik(b}, p_0, \boldsymbol{\gamma}_1)\rightarrow\boldsymbol{\zeta}_1,$
$\texttt{ik(b}, \boldsymbol{\rho}_1, \boldsymbol{\gamma}_1)\rightarrow\boldsymbol{\zeta}_2, \texttt{motion}(q_0, q_0)\rightarrow\boldsymbol{\tau}_1,$
$\texttt{motion}(q_0, \boldsymbol{\zeta}_1)\rightarrow\boldsymbol{\tau}_2, \texttt{motion}(\boldsymbol{\zeta}_1, q_0)\rightarrow\boldsymbol{\tau}_3, ...]$.

Each $\boldsymbol{\gamma}_i, \boldsymbol{\rho}_i, \boldsymbol{\zeta}_i$, and $\boldsymbol{\tau}_i$ represents a unique optimistic output. In total, 13 stream instances are created. Here, the `poses` and `grasps` stream instances are all level 1, the `ik` stream instances are all level 2, and the `motion` stream instances are all level 3. A possible optimistic plan π^*_1 and stream plan ψ_1 produced by SEARCH and RETRACE are:

$\pi^*_1 = [\texttt{move}(q_0, \boldsymbol{\tau}_2, \boldsymbol{\zeta}_1), \texttt{pick(b}, p_0, \boldsymbol{\gamma}_1, \boldsymbol{\zeta}_1), \texttt{move}(\boldsymbol{\zeta}_1, \boldsymbol{\tau}_4, \boldsymbol{\zeta}_2),$
$\texttt{place(b}, \boldsymbol{\rho}_1, \boldsymbol{\gamma}_1, \boldsymbol{\zeta}_2)]$
$\psi_1 = [\texttt{grasps(b)}\rightarrow\boldsymbol{\gamma}_1, \texttt{poses(b, r)}\rightarrow\boldsymbol{\rho}_1, \texttt{ik(b}, p_0, \boldsymbol{\gamma}_1)\rightarrow\boldsymbol{\zeta}_1$
$\texttt{ik(b}, \boldsymbol{\rho}_1, \boldsymbol{\gamma}_1)\rightarrow\boldsymbol{\zeta}_2, \texttt{motion}(q_0, \boldsymbol{\zeta}_1)\rightarrow\boldsymbol{\tau}_2, \texttt{motion}(\boldsymbol{\zeta}_1, \boldsymbol{\zeta}_2)\rightarrow\boldsymbol{\tau}_4]$

Assuming each stream evaluation is successful, the following objects are produced, which correspond to bindings $B = \{\boldsymbol{\gamma_1} : g_1, \boldsymbol{\rho_1} : p_1, \boldsymbol{\zeta_1} : q_1, \boldsymbol{\zeta_2} : q_2, \boldsymbol{\tau_2} : t_1, \boldsymbol{\tau_4} : t_2\}$. After substituting these values for their corresponding optimistic objects in π_1^*, the plan π_1 is returned as a solution.

$$\textbf{next}(\texttt{grasps(b)}) = g_1, \textbf{next}(\texttt{poses(b, r)}) = p_1$$
$$\textbf{next}(\texttt{ik(b}, p_0, g_1)) = q_1, \textbf{next}(\texttt{ik(b}, p_1, g_1)) = q_2$$
$$\textbf{next}(\texttt{motion}(q_0, q_1))) = t_1, \textbf{next}(\texttt{motion}(q_1, q_2)) = t_2$$

In the event that, for example, an inverse kinematic stream evaluation fails, *e.g.* $\textbf{next}(\texttt{ik(b}, p_0, g_1)) = \textbf{None}$, BINDING-PROCESS-STREAMS terminates, and the levels of $\texttt{grasps(b)}$ and $\texttt{ik(b}, p_0, g_1)$ are incremented to 2 and 3. As a result, both of the following optimistic stream sequences are only possible when maximum level $l \geq 4$, preventing them from being applied again until l is incremented due to SEARCH failing to find a plan ($\pi_i^* = \textbf{None}$).

1) $[\texttt{grasps(b)} \rightarrow \boldsymbol{\gamma_1}, \texttt{ik(b}, p_0, \boldsymbol{\gamma_1}) \rightarrow \boldsymbol{\zeta_1}, \texttt{motion}(q_0, \boldsymbol{\zeta_1}) \rightarrow \boldsymbol{\tau_1}]$
2) $[\texttt{ik(b}, p_0, g_1) \rightarrow \boldsymbol{\zeta_2}, \texttt{motion}(q_0, \boldsymbol{\zeta_2}) \rightarrow \boldsymbol{\tau_2}]$

8.3 Adaptive Algorithm

The *Binding* algorithm will reconsider each previously identified stream plan ψ using BINDING-PROCESS-STREAMS. However, it may perform many calls to SEARCH, each of which is expensive, before ψ can be revisted. Rather than always *explore* new optimistic plans, it may be beneficial to *exploit* our current set of optimistic plans by expending more computation to find feasible bindings for them. Doing so can be advantageous because these plans can be repeatedly processed without any overhead from SEARCH. As a result, an algorithm can *aggressively* search through the space of bindings to attempt to find a satisfying set as well as *locally optimize* for bindings that correspond to low-cost instantiations of the optimistic plan. However, there may be stream plans that are not satisfiable, such as in *Domain 2* and *Domain 3*, so an algorithm still may need to explore additional optimistic plans. This goal of *balancing* the *exploration-exploitation trade-off* (Robbins 1952) when planning optimistically is the basis for our *Adaptive* algorithm.

Instead of evaluating each stream instance only once, ADAPTIVE-PROCESS-STREAMS maintains a queue Q of bindings to repeatedly consider. Each entry contains a stream plan ψ, an optimistic plan π^*, bindings B, and the next stream plan index i to process. Q persists across all invocations and thus contains bindings for previously identified entries that can be reattempted indefinitely. On each invocation, the queue Q is processed until it is either empty or the time elapsed exceeds a timeout parameter T. The best choice of T varies per domain depending on whether it more beneficial to explore (small T) or exploit (large T). We maintain a running sum of the time spent by both SEARCH and ADAPTIVE-PROCESS-STREAMS as T_s and T_p respectively. This enables us to *adaptively* choose $T \leftarrow \max(0, T_s - T_p)$, equating the time spent by both procedures and ensuring that neither dominates the total runtime.

ADAPTIVE-PROCESS-STREAMS$(U, \psi_+, \pi_+^*; T)$:
1 $Q = [\langle \psi_+, \pi_+^*, \{\,\}, 0\rangle]$ // Initialize queue with empty binding
2 **while** $Q \neq [\,]$ **and not** TIMEOUT(T):
3 $\quad \psi, \pi^*, B, i = \text{POP}(Q)$
4 $\quad$ **if** $i = \textbf{len}(\psi)$: **return** APPLY-BINDINGS(B, π^*)
5 $\quad B' = \text{UPDATE-BINDINGS}(\textbf{copy}(B), \psi[i])$
6 $\quad$ **if** $B' \neq$ **None**: PUSH$(Q, \langle \psi, \pi^*, B', i+1\rangle)$
7 $\quad$ PUSH$(Q, \langle \psi, \pi^*, B, i\rangle)$ // Return $\langle \psi, \pi^*, B, i\rangle$ to Q
8 **return None**

Additionally, we implement Q as a priority queue that sorts entries by increasing $\textbf{count}(\psi[i])$ followed by $\textbf{len}(\psi) - i$. This approach lexicographically prefers evaluating entries with stream instances $s(\bar{x}) = \psi[i]$ that have been evaluated fewer times followed by stream plans where fewer unbound optimistic objects remain. This strategy applies the *optimism in the face of uncertainty* (Sutton and Barto 2018) principle by prioritizing partially-bound stream plans that have been explored less. Finally, we continue popping entries off of Q, despite the fact that the timeout may be exceeded, as long as $\textbf{count}(s_i(\bar{x})) = 0$ in order to greedily evaluate stream instances that have yet to be evaluated.

8.4 Rebinding

Optimistic plans may contain objects that were generated by streams. For example, in *Domain 2*, the second optimistic plan π_2^* identified (`move` actions are omitted) has the stream output objects g_1, q_1, p_1, q_2 as arguments to the `pick` and `place` for the `blue` block. Because these objects are not optimistic, they are not present as outputs in stream plan ψ_2, and thus ADAPTIVE-PROCESS-STREAMS cannot bind them.

$$\pi_2^* = [\texttt{pick(red}, p_0', \boldsymbol{\gamma_1'}, \boldsymbol{\zeta_1'}), \texttt{place(red}, \boldsymbol{\rho_1'}, \boldsymbol{\gamma_1'}, \boldsymbol{\zeta_2'}),$$
$$\texttt{pick(blue}, p_0, \underline{g_1}, \underline{q_1}), \texttt{place(blue}, \underline{p_1}, \underline{g_1}, \underline{q_2})]$$

However, the new optimistic objects $\boldsymbol{\gamma_1'}, \boldsymbol{\zeta_1'}, \boldsymbol{\rho_1'}, \boldsymbol{\zeta_2'}$ are still subject to constraints and costs involving the *fixed* objects g_1, q_1, p_1, q_2. For instance, the stream plan tail $[\texttt{motion}(\boldsymbol{\zeta_2'}, q_1) \rightarrow \boldsymbol{\tau_4}, \texttt{Dist}(\boldsymbol{\tau_4})]$ implicitly tests whether q_1 is reachable from $\boldsymbol{\zeta_2'}$ and imposes a cost based on the distance traveled along a trajectory τ_4 between them. Intuitively, we would instead want to explore combinations of all these arguments as *free* parameters. To do this, we alter line 7 in OPTIMISTIC to be $\psi = \text{RETRACE}(\underline{\mathcal{I}}, U^*, \text{PREIMAGE}(\pi + [\mathcal{G}]))$, which additionally extracts the sequence of stream instances that produced each *non-optimistic* object. As a result, fixed objects are now treated as optimistic objects that can take on new values through *rebinding*. This allows ADAPTIVE-PROCESS-STREAMS to explore additional combinations of bindings to more quickly find both feasible and low-cost solutions.

9 Experiments

We experimented using the *Incremental*, *Focused*, *Binding*, and *Adaptive* algorithms on 100 randomly-generated problems within 3 domains in section 4. The *Incremental* and *Focused* algorithms serve as baselines that are representative of prior work (Garrett, Lozano-Pérez, and Kaelbling 2018). We enforced a 2 minute timeout that includes stream

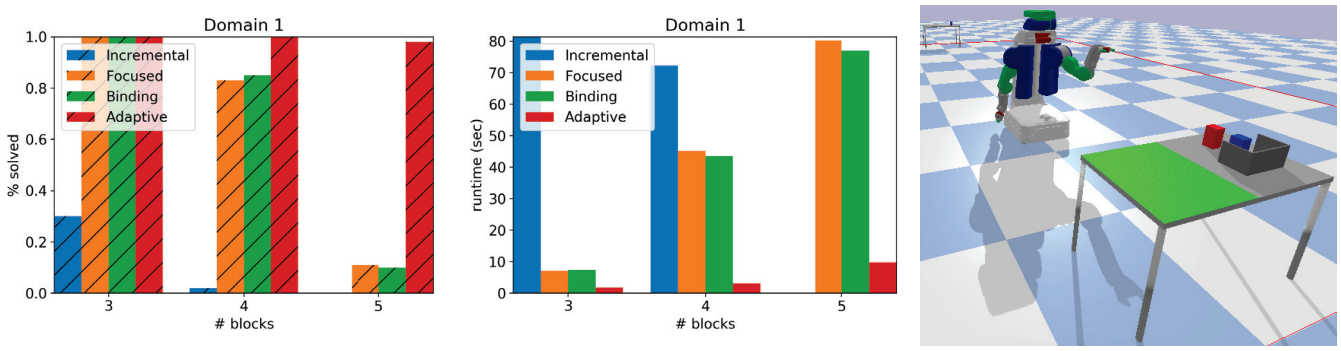

Figure 3: From left to right: *Domain 1* success percent, *Domain 1* mean runtime, and *Domain 2*.

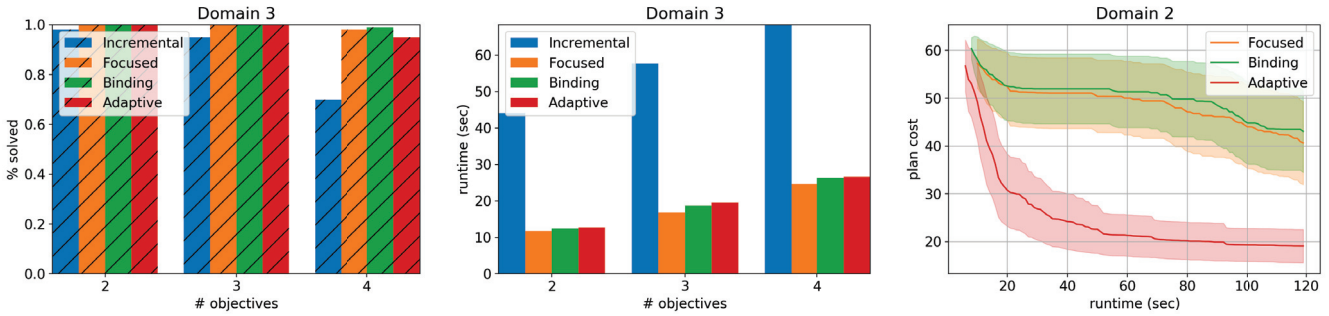

Figure 4: From left to right: *Domain 3* success percent, *Domain 3* mean runtime, and plan cost over time for *Domain 2*.

evaluation time. An open-source Python implementation is available at https://github.com/caelan/pddlstream. We use the FastDownward (Helmert 2006) planning system to implement SEARCH. The stream conditional generators were implemented using PyBullet (Coumans and Bai 2016).

Figure 4 shows the success rate and mean runtime of successful trials for *Domain 1* as the number of blocks increases from 3 to 5, which causes the problem to become more constrained. *Adaptive* outperforms *Incremental*, *Focused*, and *Binding* due to its ability to aggressively search over many bindings of a single stream plan. Figure 4 shows the average plan cost over time with a 0.5 standard deviation confidence interval for *Domain 2*. *Incremental* is omitted because it only solved 83% of the problem instances *Adaptive* converges to a low-cost solution more quickly than *Focused* and *Binding*. Figure 3 shows the success rate and mean runtime of successful trials for *Domain 3* as the number of objectives increases from 2 to 4. *Focused*, *Binding*, and *Adaptive* all outperform *Incremental* and perform about equivalently due to the less geometrically constrained nature of the domain. The additional stream binding computation only marginally increases the runtime of *Adaptive*.

9.1 Real-World Validation

We applied PDDLStream to four real-world task and motion planning problems. For each task, a PR2 robot observes the initial state, solves for a plan, and executes it in an open-loop fashion. Our PDDLStream domain description includes 9 actions: `move`, `pick`, `place`, `stack`, `push`, `press`, `pour`, `scoop`, `stir`, and `cook`. Each action is supported by one or more streams that sample its continuous control parameters. Figure 1 shows the PR2 solving the *serve* task, where it "prepares a meal" by serving a beverage (blue cup) and a cooked cabbage (green block) on the brown tray. The robot "cooks" the cabbage by placing it on the stove, turning the stove on, waiting, and turning the stove off. Like in *Domain 1*, this problem requires tightly packing the beverage and cabbage on the tray. *Adaptive* is able to quickly identify a collision-free pair of placements supporting a solution. See the extended version of this paper (https://arxiv.org/abs/1802.08705) for descriptions of the other tasks. Videos of the PR2 completing each task are available at https://tinyurl.com/pddlstream.

10 Conclusion

PDDLStream is a general-purpose framework for incorporating sampling procedures in a planning language. We introduced two new algorithms that reduce PDDLStream planning to solving a series of finite PDDL problems. Our *Adaptive* algorithm balances the time spent searching and sampling, allowing it to aggressively explore many possible bindings. As a result, it outperforms existing algorithms, particularly on tightly-constrained and cost-sensitive problems by greedily optimizing discovered plans. Finally, we demonstrated that PDDLStream can be used to plan for real-world robots operating using a diverse set of actions.

References

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1):5–33.

Bryce, D.; Gao, S.; Musliner, D. J.; and Goldman, R. P. 2015. SMT-based nonlinear PDDL+ planning. In *AAAI*.

Cashmore, M.; Fox, M.; Long, D.; and Magazzeni, D. 2016. A compilation of the full pddl+ language into smt. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 79–87. AAAI Press.

Coumans, E., and Bai, Y. 2016. Pybullet, a python module for physics simulation for games, robotics and machine learning. http://pybullet.org.

Dantam, N. T.; Kingston, Z.; Chaudhuri, S.; and Kavraki, L. E. 2016. Incremental task and motion planning: A constraint-based approach. In *Robotics: Science and Systems (RSS)*.

Della Penna, G.; Magazzeni, D.; Mercorio, F.; and Intrigila, B. 2009. Upmurphi: A tool for universal planning on pddl+ problems. In *International Conference on Automated Planning and Scheduling (ICAPS)*.

Dellin, C. M., and Srinivasa, S. S. 2016. A unifying formalism for shortest path problems with expensive edge evaluations via lazy best-first search over paths with edge selectors. *International Conference on Automated Planning and Scheduling (ICAPS)*.

Dornhege, C.; Gissler, M.; Teschner, M.; and Nebel, B. 2009a. Integrating symbolic and geometric planning for mobile manipulation. In *IEEE International Workshop on Safety, Security and Rescue Robotics*.

Dornhege, C.; Eyerich, P.; Keller, T.; Trüg, S.; Brenner, M.; and Nebel, B. 2009b. Semantic attachments for domain-independent planning systems. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 114–121. AAAI Press.

Dornhege, C. 2014. Task planning for high-level robot control.

Fernández-González, E.; Williams, B.; and Karpas, E. 2018. Scottyactivity: mixed discrete-continuous planning with convex optimization. *Journal of Artificial Intelligence Research* 62:579–664.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Fox, M., and Long, D. 2003. Pddl2.1: an extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research (JAIR)* 20:2003.

Fox, M., and Long, D. 2006. Modelling mixed discrete-continuous domains for planning. *J. Artif. Intell. Res.(JAIR)* 27:235–297.

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

Garrett, C. R.; Lozano-Pérez, T.; and Kaelbling, L. P. 2017. Ffrob: leveraging symbolic planning for efficient task and motion planning. *The International Journal of Robotics Research*.

Garrett, C. R.; Lozano-Pérez, T.; and Kaelbling, L. P. 2018. Sampling-based methods for factored task and motion planning. *The International Journal of Robotics Research*.

Gregory, P.; Long, D.; Fox, M.; and Beck, J. C. 2012. Planning modulo theories: Extending the planning paradigm. In *International Conference on Automated Planning and Scheduling (ICAPS)*.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research (JAIR)* 26:191–246.

Hertle, A.; Dornhege, C.; Keller, T.; and Nebel, B. 2012. Planning with semantic attachments: An object-oriented view. In *Proceedings of the 20th European Conference on Artificial Intelligence*, 402–407. IOS Press.

Hoffmann, J. 2003. The metric-ff planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of artificial intelligence research* 20:291–341.

Kaelbling, L. P., and Lozano-Pérez, T. 2011. Hierarchical planning in the now. In *IEEE International Conference on Robotics and Automation (ICRA)*.

Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research* 20:1–59.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. Pddl: The planning domain definition language. Technical report, Yale Center for Computational Vision and Control.

Pednault, E. P. 1989. Adl: exploring the middle ground between strips and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning*, 324–332. Morgan Kaufmann Publishers Inc.

Piotrowski, W.; Fox, M.; Long, D.; Magazzeni, D.; and Mercorio, F. 2016. Heuristic planning for pddl+ domains. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI)*.

Robbins, H. 1952. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society* 58(5):527–535.

Srivastava, S.; Fang, E.; Riano, L.; Chitnis, R.; Russell, S.; and Abbeel, P. 2014. Combined task and motion planning through an extensible planner-independent interface layer. In *IEEE International Conference on Robotics and Automation (ICRA)*.

Sutton, R. S., and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

Thiébaux, S.; Hoffmann, J.; and Nebel, B. 2005. In defense of pddl axioms. *Artificial Intelligence* 168(1-2):38–69.

Toussaint, M. 2015. Logic-geometric programming: an optimization-based approach to combined task and motion planning. In *AAAI Conference on Artificial Intelligence*, 1930–1936. AAAI Press.