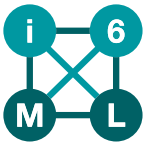


Chair of Machine Learning and Reasoning (Computer Science 6)

RWTH AACHEN UNIVERSITY

This thesis was submitted to the Chair of Machine Learning and Reasoning.

Semantic Space Abstractions for Partially Observable Deterministic Task and Motion Planning

Master Thesis

Presented by

Hani Alassiri Alhabboub
433422

Supervised by Till Hofmann, Daniel Swoboda

1st Examiner Prof. Hector Geffner, Ph.D.

2nd Examiner Prof. Christopher Morris

Aachen, June 10, 2026

Abstract

A robot acting in a cluttered scene often cannot see everything that matters: a target object may be hidden behind others, and the robot must then decide where to look as part of deciding what to do. This thesis extends classical Task and Motion Planning (TAMP) to such partial observability by integrating a *Boxel* representation into a classical TAMP planner. A *Boxel* is a box-shaped cell of the workspace. The partition gives each detected object its own cell, marks the regions those objects hide from the camera as dedicated cells, and covers the remaining open space with a few coarse cells. The robot's belief about which Boxels are occupied is encoded as Know-If Fluents (one per Boxel, marking whether its contents are known), so a classical planner can reason about hidden objects and plan sensing actions in the same search, without enumerating an exhaustive state space. On three tabletop tasks evaluated against a uniform-grid baseline at the same cell scale, the adaptive partition produces an order of magnitude fewer cells, cuts planning time accordingly, and solves cluttered scenes far more often than the uniform baseline (75.6 % vs 34.4 % success on finding a hidden cube and stacking it on a tray, 65.6 % vs 47.8 % on retrieving a hidden object). Against the POMDP-based TAMPURA framework on the closest analogous hidden-object task, the system reaches the goal at comparable end-to-end cost ($\approx$144 s vs 166 s per episode) and comparable reliability (65.6 % vs the $\geq$63 % implied by its reported returns), without learning a probabilistic policy.

Contents

1 Introduction

We want robots that can carry out everyday tasks on their own, like fetching an object or clearing a table. Such tasks demand both multi-step reasoning and careful physical manipulation. Task and motion planning (TAMP) studies how to do exactly this [9, 12, 13]. TAMP connects a high-level goal (e.g., "fetch the blue cup from the kitchen") to the actual motions the robot must perform to achieve it. TAMP combines discrete task planning with continuous motion planning. Executing even a simple instruction like (*pick A*) requires more than the symbolic step: object A must be reachable, a grasp for it must exist, and a collision-free motion must carry the arm to it. A symbolic planner generates a high-level sequence of abstract actions (the *task plan*) while ignoring this geometry, so the plan can call for actions the robot cannot physically carry out. A TAMP system therefore checks each proposed action against the robot's continuous geometric and kinematic constraints and fills in the concrete parameters needed to execute it, such as object poses, grasp configurations, and collision-free trajectories.

One widely used framework for this is PDDLStream [13]. PDDLStream extends the classical Planning Domain Definition Language (PDDL) [31] by incorporating streams: lazily-evaluated, declarative procedures that can interface with external solvers. These streams are capable of generating or validating continuous parameters on demand, allowing the symbolic planner to retrieve geometric or kinematic information only as needed. This simplifies the coupling between high-level symbolic reasoning (what to do) and low-level geometric feasibility (how to do it), making the planning process both flexible and efficient.

However, the original PDDLStream framework assumes full observability of the world state and deterministic action outcomes, assumptions that rarely hold in practice. Robots frequently operate under partial observability, due to occlusions, limited sensor coverage, or dynamically changing environments. In such settings, crucial information, such as the location of a target object or the presence of an obstacle, may be unknown at the start, so the robot has to actively look for it while carrying out the task (Figure 1.1).

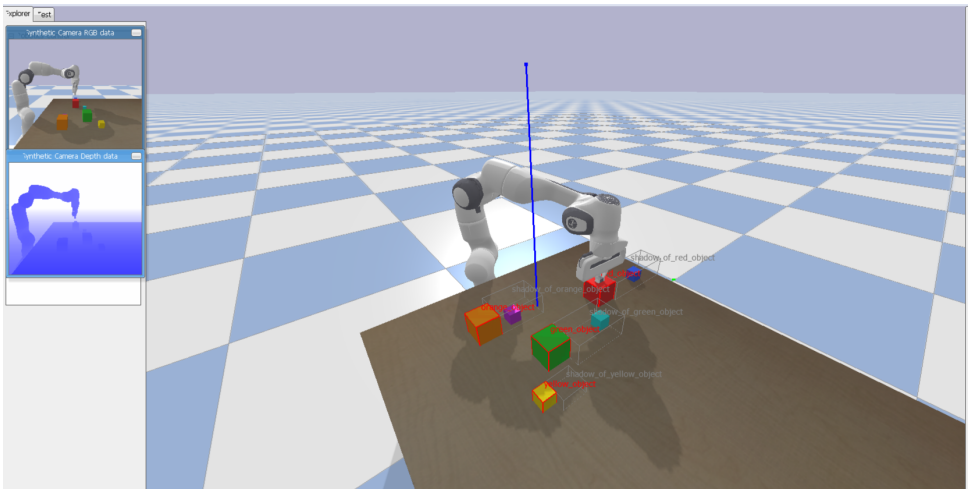


Figure 1.1: A 7-DOF Franka Panda at a cluttered tabletop. Red wireframe boxes mark the object Boxels bounding each detected cube; the grey regions behind them are shadow Boxels, the volumes those objects hide from the camera. The target, the cyan cube, lies inside the shadow region of the green cube and is therefore hidden from the fixed camera; it must be located before it can be retrieved. The corner panels are the fixed camera's synthetic RGB (top) and depth (bottom) views.

Addressing this limitation is the central focus of this thesis: extending PDDLStream to operate under partial observability. This includes the ability to represent and reason over what the robot knows and does not know about object locations: a discrete belief over spatial regions, rather than a full probabilistic belief over continuous variables. Naive representations, such as a uniform voxel grid over 3D space, make the belief space blow up in size and quickly become impractical to plan with. A useful belief representation must therefore be detailed enough to be expressive, yet small enough to plan with quickly. The core research question is: *How can we formally define and integrate a belief representation based on semantic partitioning into a partially observable deterministic (POD) TAMP framework?* The stance of this thesis is that the structure of space should enter the symbolic planning state itself: where a hidden object might be, and what blocks the view of it, are not details to be collapsed into a probabilistic occupancy grid but facts the planner should weigh directly. Here, we instead present a novel approach to discretizing the workspace around the objects the robot detects, the regions they occlude, and the free space between them. The unit of this discretization is the Boxel: a task-relevant, axis-aligned box that the robot uses to represent its belief. A Boxel either bounds a detected object, marks a region an object hides from the camera, or covers a region of free space; the object and shadow cells are constructed from the objects' known 3D models. Unlike a dense, uniform discretization, this provides a structured, scalable way to represent and reason about spatial uncertainty.

This thesis develops a system[1] in which a robot, facing uncertainty about objects, can plan and execute a sequence of information-gathering and manipulation actions. It reasons over abstract beliefs about which Boxels objects might occupy, using PDDLStream to connect this reasoning to continuous motion and manipulation. Sensing is a symbolic action evaluated against a single fixed camera, not a viewpoint the robot moves to; and because the underlying planner is classical, partial observability is handled by optimistic sensing with reactive replanning rather than contingent planning: each sensing action is assumed to find the target it looks for, and the system replans if execution shows otherwise. The result is a framework that represents spatial uncertainty compactly and plans over it quickly and reliably. The approach is demonstrated on tabletop hidden-object retrieval and stacking; the partial-observability mechanism is the main contribution.

1.1 Contributions

The central contribution is a *representation*: each volume the robot cannot see becomes a named region that a classical planner reasons over and resolves by sensing. The three components below deliver this representation; the evaluation correspondingly isolates it by ablation (the same planner on the adaptive partition against a uniform grid), rather than claiming a benchmark win, and is conducted under oracle perception (Section 5.5).

- The *Boxel* abstraction: an occlusion-aware, object-centric partitioning of the workspace that concentrates resolution on the task-relevant regions (the objects themselves and the volumes they occlude) while covering the remaining free space with a few coarse cells, rather than partitioning all space uniformly.
- A POD-TAMP model that uses Know-If Fluents over Boxels (atoms marking whether the truth of a fact is known) as a compact belief representation, letting a classical PDDL planner reason about partial observability via optimistic sensing with reactive replanning.
- A PDDLStream realization of this model that combines the discrete Boxel belief with stream-certified continuous geometry, so that information-gathering and manipulation actions interleave within a single planning domain.

1.2 Outline

Chapter 2 provides essential background on classical planning, reasoning under uncertainty, and the PDDLStream framework that this work builds upon. Chapter 3 reviews related work in spatial representations for planning under occlusion and in TAMP under partial observability, belief representation, and adjacent planning paradigms. Chapter 4 details the methodology, including the formalization of Boxels through adaptive semantic discretization, the POD-

[1] Source code: `https://git.rwth-aachen.de/hani.alassiri.alhabboub/Semantic_Space_Abstractions`.

TAMP model, and the PDDLStream integration. Chapter 5 presents the experimental setup, the metrics and baselines, and the evaluation results, interprets them, and consolidates the accepted simplifications. Chapter 6 summarizes the contributions and outlines directions for future work.

2 Background

This chapter covers the four ingredients the approach builds on: classical planning and the algorithms that solve it, planning under partial observability, the PDDLStream framework, and spatial representations of 3D space.

2.1 AI Planning Fundamentals

Artificial intelligence (AI) planning produces a sequence of actions, a plan, that gets from a given initial state to a specified goal [16].

2.1.1 State-Space Models in Classical Planning

Classical planning describes a problem in terms of small facts called *atoms*: individual true-or-false statements about the world, such as "block A is on block B" or "the gripper is empty". A state is then given by the set of atoms that currently hold, and an action is described by the atoms it makes true or false. Describing the world this way avoids enumerating its possible configurations explicitly, which is infeasible for all but the smallest problems. This compact, *factored* representation is what keeps planning tractable [15, 16].

STRIPS [11] is a common language to express such factored planning problems. A **state** is the set of ground atoms (propositions) true in a configuration of the world. For example, in a blocksworld with blocks A, B and a table, a state is {`(on A B)`, `(ontable B)`, `(handempty)`}. An **action** a has preconditions $\mathrm{pre}(a)$, positive effects $\mathrm{add}(a)$, and negative effects $\mathrm{del}(a)$, each expressed as a set of atoms.

PDDL [31] gives this a standardized, more expressive syntax. A planning problem is a tuple $P = \langle D, I \rangle$ consisting of a domain D and an instance I. The **domain** is a tuple $D = \langle \mathcal{P}, \mathcal{A} \rangle$, where $\mathcal{P}$ is a set of *predicates* (relations such as `(on ?x ?y)`) and $\mathcal{A}$ is a set of *action schemas*, parameterized templates that generalize STRIPS operators, e.g.:

```
(:action stack
    :parameters (?obj1 - block ?obj2 - block)
    :precondition (and (holding ?obj1) (clear ?obj2))
    :effect (and (on ?obj1 ?obj2) (not (holding ?obj1))
                 (not (clear ?obj2)) (handempty) (clear ?obj1))
)
```

whose negative effects (`(holding ?obj1)`, `(clear ?obj2)`) are its del(a) and whose positive effects are its add(a). The **instance** is a tuple $I = \langle \mathcal{O}, s_0, g \rangle$, where $\mathcal{O}$ is a set of *objects*, s_0 is the *initial state* (the ground atoms true at the start), and the *goal* g is a set of ground atoms (those required in any goal state). Ground atoms are obtained from $\mathcal{P}$ by substituting each predicate's arguments by objects from $\mathcal{O}$; ground actions result from grounding the action schemas by replacing all schema arguments by objects from $\mathcal{O}$. PDDL also supports features beyond STRIPS, such as conditional effects (`(when <condition> <effect>)`) and axioms (derived predicates).

A planning problem P defines a *state model* $S(P) = \langle S, s_0, S_G, Act, A, f \rangle$ as its semantics [15, 24]: S is the set of all states; $s_0 \in S$ the initial state; Act the set of ground actions; $A(s) = \{ a \in Act : \text{pre}(a) \subseteq s \}$ the actions applicable in s; $f(a, s) = (s \setminus \text{del}(a)) \cup \text{add}(a)$ the successor state for $a \in A(s)$; and $S_G = \{ s \in S : g \subseteq s \}$ the goal states. A *solution* (or *plan*) is a sequence of actions $\sigma = [a_0, \dots, a_n]$ with $a_i \in A(s_i)$ and $s_{i+1} = f(a_i, s_i)$ that ends in a goal state $s_{n+1} \in S_G$; an *optimal* plan minimizes length or an associated cost.

2.1.2 Classical Planning Algorithms and Fast Downward

Given a compact PDDL encoding, a planner must find a path from the start state s_0 to a goal state through the space of states the encoding implies. The difficulty is sheer size: every atom that can independently be true or false doubles the number of possible states, so even modest problems have astronomically many, and they cannot be enumerated. Modern classical planners instead use *heuristic search*. The search explores states one at a time, and at each step a *heuristic* (a fast, approximate estimate of how many actions still separate a state from the goal) tells it which state looks most promising to expand next. Greedy best-first search simply expands whichever state the heuristic rates closest to the goal, trading guaranteed optimality for speed; A* [20] instead weighs that estimate against the cost already incurred and returns a provably shortest plan whenever the heuristic never overestimates [15]. In practice a planner is only as good as its heuristic.

What makes such planners *domain-independent* (able to tackle any problem written in PDDL without hand-tuning) is that the heuristic is derived automatically from the problem description. The most influential idea is the *delete relaxation*: pretend that actions only ever add facts and never remove them, so nothing already achieved can be undone. This relaxed problem is much easier to solve, and the effort it takes is a good estimate of the true distance to the goal; the max, additive, and FF heuristics are practical ways to compute that estimate, the FF heuristic by extracting a plan for the relaxed problem and counting its actions [15, 23]. A complementary idea is *landmarks* (facts that every valid plan must make true at some point), whose count gives another distance estimate [38].

Fast Downward [21] is the classical planner used as the search backend throughout this work; the PDDLStream framework (Section 2.3) invokes it to solve each symbolic planning problem once its streams have supplied the continuous values that problem requires. Its distinctive step is a change of representation. A STRIPS state is a long list of independent true/false facts, but many of those facts are really mutually exclusive: a given block sits on exactly one thing at a time. Fast Downward detects such groups automatically and rewrites the task in a *multi-valued* form (SAS$^+$), replacing a cluster of related Booleans with a single variable that takes one value from a fixed set, for example `pos(A)` $\in$ {`onB`, `ontable`, `inhand`}. This both shrinks the problem and reveals its structure: Fast Downward builds a *causal graph* recording which variables influence which others, and derives structure-aware heuristics (such as the causal-graph heuristic) that exploit it to guide a best-first search. Pairing this automatic multi-valued translation with structure-exploiting heuristics helps Fast Downward handle the grounded planning tasks that arise in this work.

2.2 Planning under Partial Observability

Classical planning's assumption of full observability is often violated in robotics. When an agent has incomplete information about the true state of the world, it must reason about its *belief state*.

2.2.1 Belief States, K-Literals, and Know-If Fluents

A belief state represents the agent's current set of possible world states consistent with its history of actions and observations. We refer to the setting this thesis targets as *Partially Observable Deterministic (POD)* planning: the world itself behaves deterministically and the only uncertainty is the agent's incomplete knowledge of the initial situation, which sensing resolves. This deterministic, knowledge-based view of partial observability draws on a line of compilation-based planners [1, 6, 15]. The belief state b is explicitly a set of possible states. Actions transition this set, and observations prune it. A common way to reason about knowledge in POD planning is through K-literals [5, 15]. K-literals make the robot's knowledge about a proposition p explicit. For example:

- $K(p)$ signifies "p is known to be true."
- $K(\neg p)$ signifies "p is known to be false."

If neither $K(p)$ nor $K(\neg p)$ holds, the truth of p is unknown. Separately, p is "possibly true" whenever $K(\neg p)$ does not hold, which also covers the case where p is known to be true.

K-literals are the basis of a *translation* that compiles a partially observable problem into a classical one [15]. The translation replaces each literal of the original problem with a knowledge literal: L becomes $K(L)$ ("L is known true") and $\neg L$ becomes $K(\neg L)$ ("L is known false") [35],

so every proposition contributes two independent fluents. The two are jointly consistent but not exhaustive: at most one is ever true (asserting both would be contradictory knowledge), and an unknown proposition has both false. In the initial state each is set true only where the agent actually knows the corresponding value, and an unknown proposition contributes neither. The translated problem therefore carries no uncertainty in its initial state and can be solved by an ordinary classical planner, which now searches for a plan that reaches a desired *state of knowledge* (for example K(holding(target_obj)), i.e. the agent knows it holds the target) rather than a desired world state. This thesis instead carries belief with Know-If Fluents (KIFs), the know-whether construct of the PKS planner of Petrick and Bacchus [37], also used in the continual planning of Brenner and Nebel [7]. A Know-If fluent is weaker than a K-literal: rather than asserting that a fact is known true ($K(L)$) or known false ($K(\neg L)$), it records only that the agent knows *whether* the fact holds (that its value is settled) while the value itself is carried by an ordinary fluent. The two are therefore not interchangeable: $K(L)$ commits to L being true, whereas a KIF states only that L's value is known and leaves that value to the underlying fact. We adopt KIFs because the domain already stores where each object is, so it suffices to add one Know-If fluent per fact (whether that location is known) rather than two separate knowledge literals $K(L)$ and $K(\neg L)$ (Section 4.2).

2.2.2 Probabilistic vs. Deterministic Partial Observability

Where a classical planner assumes each action has a single, certain outcome, a Markov decision process (MDP) [15] models actions whose outcomes are *stochastic*. It is defined by a set of states, a set of actions, a transition function giving the probability of each next state, and a reward at each step. Its solution is not a fixed action sequence but a *policy*, a choice of action in every state, that maximises expected cumulative reward; outcomes depend only on the current state and action, not on how it was reached (the *Markov* property). A common example is a robot crossing a grid whose moves occasionally slip sideways, so the same action can land it in different cells.

A partially observable Markov decision process (POMDP) [26] adds partial observability: the agent cannot see the true state but only receives noisy observations, so it acts on a *belief state*, a probability distribution over possible states. POMDPs are expressive but computationally very demanding. POD planning keeps the belief but drops the randomness: outcomes and observations are deterministic, and the only uncertainty is incomplete knowledge of an otherwise fixed world. This makes it more tractable than a POMDP, and it fits robotic problems where the uncertainty comes from not knowing the world rather than from actions behaving randomly.

2.3 Task and Motion Planning (TAMP)

TAMP connects high-level task goals with a robot's actual physical movements [12]. The core problem is ensuring that a plan is physically possible. A high-level planner works on an abstraction that leaves geometry out, so a plan that is symbolically correct may not be executable. Yet a planning model that carries full geometric detail is too complex to solve directly. Combining high-level logic with real-world geometry is the central difficulty of TAMP, and frameworks like PDDLStream exist to bridge it.

2.3.1 PDDLStream for TAMP

PDDLStream [13] is a TAMP framework built to bridge this symbolic-continuous gap. It extends classical PDDL by introducing the core concept of **streams**.

Streams: declarative procedures for continuous parameters. A stream in PDDLStream is a procedure that generates or tests the continuous values symbolic actions require, such as object poses, robot configurations, grasp transformations, or motion trajectories. A stream does not only fill in parameter values: each output it produces is itself a new object that the symbolic layer can name, so streams also introduce the existence and identity of the continuous objects the planner reasons about. Streams have both a procedural and a declarative component:

- **Procedural Component (Conditional Generator):** This is the actual code (e.g., a Python function) that, given some input values (which can be discrete symbols or other continuous values), produces a (potentially infinite) sequence of output values. For example, an inverse kinematics (IK) stream, given an object, its target pose, and a grasp, produces robot configurations that achieve that end-effector pose. A single pose usually admits many valid configurations, so the stream can supply several: the planner takes one and, if it does not work out (say the arm would collide), asks the stream for another. The outputs are a pool of interchangeable candidates the planner draws from, not a fixed order of steps. A collision-checking stream, given a trajectory, would test if it is collision-free.
- **Declarative Component (Stream Specification):** This part is described in a PDDL-like syntax and informs the symbolic planner what the stream does, what inputs it requires, what outputs it produces, and, crucially, what logical conditions (fluents) are associated with these inputs and outputs. This specification includes:
  - `:inputs`: The typed parameters the stream procedure takes.
  - `:domain`: A set of PDDL facts (preconditions) that must be true for the stream to be applicable for a given set of inputs. For example, an IK stream might require that a valid pose and grasp for the object are already known.
  - `:outputs`: The typed parameters that the stream procedure generates.

- `:certified`: A set of PDDL facts that are guaranteed to be true if the stream successfully produces a set of outputs for given inputs. These "certified fluents" are added to the planner's state and represent new knowledge derived from the continuous domain. For instance, if an IK stream successfully finds a configuration `?q` for picking object `?obj` at pose `?p` with grasp `?g`, it might certify the fluent `(KinSolution ?obj ?p ?g ?q)`.

PDDLStream employs a **lazy evaluation** strategy. Streams are not called exhaustively beforehand; instead, the symbolic planner calls them "on-demand" when it encounters an action whose preconditions require a continuous parameter to be instantiated or a continuous condition to be tested.

Example PDDLStream workflow. Consider a symbolic action `(pick ?obj ?p ?g ?q ?t)` (pick object `?obj` from pose `?p` using grasp `?g` via configuration `?q` along trajectory `?t`). For this action to be applicable:

1. A stream might first be called to sample a stable `?p` for `?obj` on a surface. If successful, it certifies `(AtPose ?obj ?p)`.
2. Then, another stream samples a grasp `?g` for `?obj`. If successful, it certifies `(Grasp ?obj ?g)`.
3. An IK stream then takes `?obj, ?p, ?g` as input and tries to find a robot configuration `?q`. If successful, it certifies `(KinSolution ?obj ?p ?g ?q)`.
4. Finally, a motion planning stream takes the current configuration and `?q` as input to find a collision-free trajectory `?t`, certifying `(Motion ?q_start ?t ?q)`.

Only if all these streams succeed and certify their respective fluents will the preconditions of the symbolic `(pick ...)` action be met.

The overall PDDLStream algorithm reduces a TAMP problem to a sequence of finite PDDL problems, iteratively calling streams to discover more certified fluents until a complete, grounded plan is found. The original PDDLStream framework, however, was primarily designed for fully observable and deterministic settings.

2.4 Spatial Representation for Robotic Planning

A robot needs some way to represent the 3D space around it and the objects in it. Representing space is harder under partial observability, because the robot must also represent what it does not yet know.

2.4.1 Discretization of Continuous Space

Robots operate in continuous space, but symbolic planners reason over discrete states, so the continuous space has to be discretized somehow.

Voxel grids. A common approach in robotic mapping, though not in symbolic planning, is to discretize the 3D environment into a grid of uniform volumetric pixels, or **voxels**. Each voxel can store information, such as whether it is occupied, free, or unknown. While straightforward, uniform voxel grids can be memory-intensive and computationally expensive for large environments or high resolutions.

Octrees (Figure 2.1) [25] offer a more efficient hierarchical data structure for representing 3D space. An octree is a tree data structure in which each internal (non-leaf) node has exactly eight children, while leaf nodes have none. It starts with a single large voxel encompassing the entire space of interest. If this voxel is not homogeneous (e.g., it contains both free and occupied space), it is recursively subdivided into eight smaller child voxels (octants).

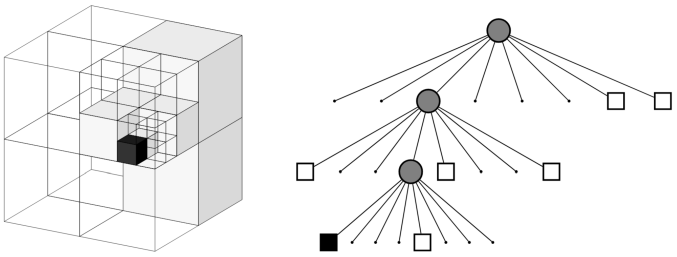


Figure 2.1: Example of an octree storing free (shaded white) and occupied (black) cells. The volumetric model is shown on the left and the corresponding tree representation on the right. Figure reproduced from Hornung et al. [25].

This subdivision continues until voxels are homogeneous or a minimum resolution/maximum depth is reached. This process creates a tree where large, uniform regions (whether free, occupied, or unknown) are represented by single nodes at a shallower depth, making it highly efficient for representing sparse environments in comparison to uniform grid voxelization. The **OctoMap** framework [25] uses this principle to build probabilistic 3D occupancy maps from noisy sensor data. In such a framework, unobservable space is typically represented implicitly by the absence of nodes in a given volume; querying such a region returns an “unknown” status. The same octree structure also underpins learning-based 3D methods such as OctNet [39], which runs deep convolutions efficiently on an octree rather than building an occupancy map; here it is the occupancy-mapping use that is relevant.

Critical Regions. Rather than dividing space by uniform geometric criteria, space can be divided by task-relevant meaning, so the planner reasons over a few salient regions instead of

many uniform cells. One such approach is the concept of **critical regions (CRs)**, introduced by Molina et al. [32] and later extended by Shah and Srivastava [42]. In the original formulation, a critical region is a part of the configuration space with a low probability of being sampled by a uniform sampler yet essential to a solution: narrow passages such as doorways [32]. Shah and Srivastava generalised this to regions where task-relevant features (controller success rates, object visibility, or transition likelihoods) change significantly [42].

In this line of work the regions are not hand-specified: a learned model *predicts* the critical regions of a given environment from that scene's input, and the predictor generalises to environments it was not trained on. Once predicted, a region-based Voronoi diagram (RBVD) partitions the configuration space around the CRs: each point is assigned to the nearest critical region, forming abstract states. Transitions between these abstract states define abstract actions, forming the backbone for hierarchical planning. By focusing computation on these task-relevant regions, this approach scales planning more effectively.

3 Related Work

Integrating task and motion planning with partial observability is challenging, particularly when a robot must act on incomplete information about its environment. This chapter reviews previous work along the two axes the thesis combines: spatial representations a planner can reason over (Section 3.1), the axis on which this thesis contributes, and planning approaches for TAMP under partial observability (Section 3.2).

3.1 Spatial Representations for Planning under Occlusion

The contribution of this thesis is a spatial representation, so the closest related work is not only other planners but other ways of carving up space. The standard occupancy representations, uniform voxel grids [10] and octree occupancy maps such as OctoMap [25] (Section 2.4), answer where matter is; unobserved space appears only implicitly, as the absence of evidence. For planning to *find* hidden objects this occupancy focus is the structures' main limitation: their subdivision is purely geometric, driven by occupancy rather than task meaning, so nothing in them names which unknown volume matters to the task, which object hides it, or what must move for it to become visible; a planner searching for a hidden object would have to infer that meaning from the raw geometry, a hard problem in its own right. The octree itself reappears inside our method, as the subdivision stage of the adaptive free-space partition (Chapter 4).

A second line, learned critical regions [32, 42], abstracts space by task relevance instead of occupancy (Section 2.4). In this formulation the regions are predicted once per environment and then held fixed, so they do not adapt as objects move within a scene, and the abstraction carries no notion of occlusion; for finding hidden objects this matters directly, since the relevant region, "behind object A", depends on where A currently is and where the camera stands. Later work in the same line learns abstractions that adapt as the task changes [34], though for reinforcement learning rather than TAMP. Boxels keep the task-relevance idea but trade the learned region predictor for geometry: regions constructed from the detected objects and their sightlines to the camera, carrying occlusion explicitly, rebuilt whenever the scene changes.

Planning systems that do reason about occlusion represent it as a property of a probabilistic belief rather than as a region of space. TAMPURA's hidden-object benchmark keeps a visibility voxel grid that weights the success probability of its `look` action [9]; IBSP carries maximum-likelihood occlusion fluents [18]; SS-Replan derives an occlusion predicate by ray-casting over its pose belief [14]; and occlusion-aware retrieval searches the likely poses of a hidden target under a learned distribution [3]. In each case occlusion is a check or a weight computed against

a distribution over object poses, not a named occluded volume that a planner can sense, clear, or avoid re-blocking as symbolic state. That named-volume representation, constructed geometrically and held as deterministic planning state, is what the Boxel partition adds; Section 3.2 reviews the planning frameworks it is evaluated against.

3.2 TAMP under Partial Observability

Existing approaches to TAMP under partial observability generally either use formal, probabilistic models or more ad-hoc, reactive methods, or a combination of both.

3.2.1 POMDP-based TAMP Solutions

Several works formulate planning under uncertainty as a POMDP to handle it in a principled, risk-aware way [9, 41]. TAMP with Uncertainty and Risk Awareness (TAMPURA) [9], for example, learns a coarse abstract model of each given controller's preconditions and effects, builds a non-deterministic MDP, and solves it with loop AO* search (LAO*), an uncertainty-aware solver [19], to produce risk-aware, information-gathering plans. [41], by contrast, addresses contact-rich *manipulation* under object-pose uncertainty (not full task-and-motion planning), computing partial policies with an anytime heuristic-search POMDP solver. However, a persistent challenge is the representation of belief over continuous properties, particularly object poses. TAMPURA [9] samples object placements in continuous pose space and, in its real-robot pipeline, draws its object-pose belief from a perception front-end, Bayes3D [17]: a generative inverse-graphics model that infers a posterior over 3D scenes by rendering candidate object arrangements and scoring them against the observed depth image.

3.2.2 Partially Grounded Plans in TAMP

In contrast to formal probabilistic models, other TAMP systems handle uncertainty by deferring it to execution [36]. These methods generate partially grounded plans, leaving "gaps" for actions with unknown outcomes. During execution, specialized closed-loop behaviors (hand-designed or learned) attempt to fill these gaps. If a controller fails, the failure is treated as a new constraint, and the system replans. This reactive approach avoids the need to model outcome probabilities explicitly. However, because it does not reason about uncertainty proactively, it cannot plan multi-step information gathering: sensing never appears as a step in the plan.

3.2.3 Modern Approaches to TAMP under Uncertainty

Recent works have leveraged learning and large models to address partial observability, offering different strategies than our own.

[30] adds a strategic reasoning layer on top of a standard planner: it first reasons backward from an underspecified goal to decide which objects matter (Task-level Backward Search), then uses

a constraint graph to order object moves collision-free through clutter (Object Manipulation Constraint Graph). We differ at the representation level rather than by adding a layer: we introduce a geometric belief representation, Boxels, so that the planner treats which regions are observed versus occluded as first-class planning state and information-gathering becomes a core planner action rather than a separate reasoning stage; the visibility signal is currently supplied by an oracle.

Another approach sources the planner's belief from large foundation models. [43] uses a vision-language model (VLM) as an uncertainty estimator: the VLM answers the planner's perceptual queries about an image (for example, "Is the drawer empty?"), and the resulting belief feeds a *symbolic* belief-space planner, a pairing of learned perception with structured reasoning that is reported to outperform end-to-end VLM planners. CoCo-TAMP [28] is similar in spirit, drawing on the commonsense priors of a large language model (LLM) to shape the belief over where task-relevant objects are likely to be, again within a partially observable TAMP loop. The difference from our work is the *source* of the belief: theirs is learned (a VLM or LLM), whereas ours is a structured, object-centric geometric partition into discrete Boxels, with object poses currently supplied by a ground-truth oracle rather than a learned detector. Integrating a real detector is left to future work.

Finally, some methods move away from symbolic planning altogether and instead use end-to-end learning. [2] (TAVP), for example, uses a separate reinforcement-learning policy to select task-relevant *virtual* camera viewpoints and re-render the scene from them, rather than physically moving the camera, and then produces the manipulation action with an end-to-end learned policy rather than a planner. This is fundamentally different from our model-based approach, where looking around is not a learned reflex but an explicit action chosen by the planner because its world model indicates that knowledge is missing. Because our method separates planning from execution, it is more transparent and can be retargeted to new goals without retraining a policy; the cost, however, is that a genuinely new goal requires authoring new PDDL actions and their streams, not just a new goal specification.

3.2.4 Belief-Space Planning and Replanning

A long line of work plans directly in *belief space*, treating the robot's distribution over world states as the object of planning. [27] plans over the robot's belief using hierarchical goal regression, while [18] introduces a modular interface that exchanges information between an off-the-shelf classical planner and the geometric layer as the plan is built, determinising the problem under the maximum-likelihood-observation assumption so that a successful plan generates the observations it needs. A practical alternative to maintaining an explicit stochastic model is to *determinise and replan*: SS-Replan [14] plans deterministically in a hybrid belief space and replans whenever a new observation arrives, including sensing actions within a single plan. Our system shares this determinise-and-replan strategy: we commit to an optimistic

deterministic plan and replan on each observation rather than synthesising a probabilistic policy. What we add is separate from the replanning loop: a spatial belief in which occlusion is an explicit, named planning state, so information-gathering is explicitly composed within the planning representation.

3.2.5 Partially Observable Deterministic Planning

A separate tradition handles partial observability *without* probabilities, by reasoning about what is *known*. A prominent line within partially observable deterministic (POD) planning represents knowledge with explicit *knowledge literals* (Section 2.2) and reduces the problem to classical planning by compilation; other POD planners instead search belief space directly, as in Contingent-FF [22] and MBP [4]. The translation-based contingent planner CLG [1] and the classical-replanning K-replanner [5] established this reduction; LW1 [6] made it scalable with a *linear* translation; and a parallel route compiles partial observability to fully-observable non-deterministic (FOND) planning [33]. This thesis builds directly on the POD line: we carry Know-If Fluents (e.g. whether a region's contents are known) inside the PDDL state and let a classical planner reason over them.

To our knowledge, the deterministic, knowledge-literal POD line has not previously been combined with task and motion planning, nor used a geometric belief representation that makes *which regions are occluded* part of the symbolic state. The systems that do reason about occlusion do so over probabilistic beliefs or learned models (Section 3.1), and partially observable TAMP systems increasingly source their belief from learned models [28, 43] or probabilistic abstractions [9]; we are not aware of prior work that represents occluded volumes as named, object-centric symbolic state resolved by sensing within a PDDLStream TAMP loop, the gap this thesis addresses.

3.3 Summary and Research Gap

These approaches occupy distinct points in the design space. POMDP-based methods such as TAMPURA [9] reason about uncertainty proactively but rely on a learned probabilistic transition model, and they keep the spatial belief sub-symbolic, so occluded regions are never a named, inspectable state that the planner can target. Reactive, partially-grounded methods [36] avoid probabilistic modelling but cannot plan multi-step information-gathering. Belief-space replanning [14] determinises and replans, as we do, but does not make occlusion a first-class planning entity. The POD tradition [6] reasons about knowledge symbolically and deterministically, but has not been applied to TAMP or to spatial occlusion. We build on these lines: a POD task-and-motion planner whose spatial belief is an adaptive, object-centric partition (Boxels) in which occluded regions and their observed/occluded status are an explicit symbolic state. The planner thus reasons about *where* information is missing and composes

deliberate look-and-clear actions, without the cost and opacity of a learned probabilistic model. On the spatial side, Section 3.1 positions the partition against the existing representations: occupancy structures carry no semantic notion of an occluded volume, learned task-relevance abstractions are predicted once per scene rather than reconstructed as objects move, and occlusion-aware planners keep occlusion inside a probabilistic belief. None names the occluded volumes that sensing must resolve, which is what the Boxel partition adds.

This position yields three advantages, each with a limitation we state plainly. (i) *Occlusion is an inspectable state*: named observed-versus-occluded regions a reader can enumerate, rather than a sub-symbolic visibility grid feeding a learned outcome model [9]; the limitation is that the visibility signal is currently supplied by an oracle rather than a perception module. (ii) *Uncertainty is handled without probabilities*: Know-If Fluents record what is known deterministically, with no per-action probability to estimate; the limitation is that the planner cannot rank actions by success likelihood and instead minimises plan cost and replans when an optimistic assumption fails. (iii) *No MDP machinery is required*: a lightweight deterministic PDDL layer replaces the learned-MDP stack of POMDP-based TAMP [9]; the limitation is that optimistic determinise-and-replan is sound only while belief reflects observation.

4 Methods

This chapter presents our approach to bridging a discrete belief over workspace regions and continuous geometry: a spatial abstraction called Boxels that partitions the robot's workspace into meaningful regions. We integrate these into a POD-TAMP framework through three components: (1) an adaptive discretization that builds Boxels from detected objects and their occlusions, (2) a formal model that represents belief about Boxel occupancy as Know-If fluents, and (3) a PDDLStream implementation that grounds these facts in geometry and sensing. On the continuous side, manipulation uses sampled top-down grasps, an inverse-kinematics solver, and a sampling-based (RRT) motion planner, while perception uses ray-casts from the fixed camera against an oracle pose detector; an outer sense–plan–act loop (Algorithm 2) ties planning, execution, and belief update together.

4.1 Adaptive Semantic Discretization: Generating Boxels

A *Boxel* is a box-shaped cell of the workspace; we build the Boxel abstraction through a process of *adaptive semantic discretization*. This process dynamically generates a set of Boxels ($\mathcal{BX}$) by partitioning the workspace based on detected objects and the occlusions they create, and is re-run as the scene changes so the partition stays current. Figure 4.1 illustrates the process, and Figure 4.2 shows the resulting partition on a simulated PyBullet scene.

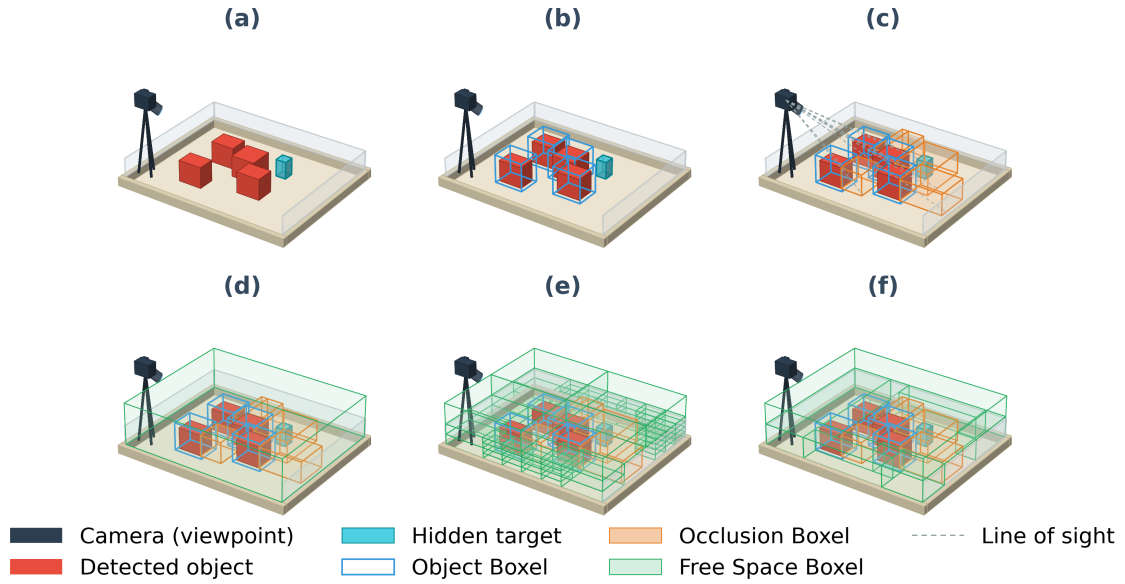


Figure 4.1: Adaptive Semantic Discretization. (a) Initial scene: the camera viewpoint and the detected objects on the table, with a fully occluded target (cyan) sitting in the shadow behind the largest occluder. (b) Each object is bounded by a dedicated object Boxel. (c) The volume each object hides from the camera becomes an occlusion (shadow) Boxel, swept from the object's back face along the lines of sight (dashed) to the far end of the region it occludes, where those sightlines meet the table behind it, and rising to the object's height. (d)–(f) The remaining free space is partitioned by an octree: from a single cell spanning the whole workspace (d), only cells containing both free and occupied space are recursively subdivided, in all three dimensions, so the open space above the objects stays a few large cells (e), and adjacent free cells are then greedily merged across aligned faces into larger convex Free Space Boxels (f).

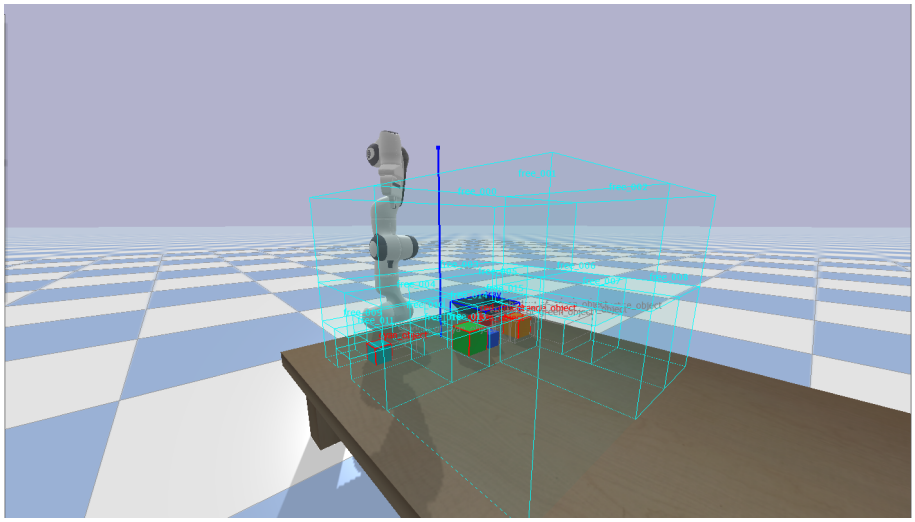


Figure 4.2: A screenshot of the same partition in the PyBullet simulator; the text labels overlaid in the scene name each object and the shadow region it casts. The cyan wireframe cells are the Free Space Boxels filling the workspace; at the cube cluster, the red wireframe cells are the object Boxels enclosing the cubes and the grey wireframe (shadow) Boxels mark the volumes they occlude.

The generation process is as follows (Algorithm 1):

1. **Object-Centric Bounding:** When an object is detected, we generate a dedicated Boxel: an axis-aligned box that tightly bounds the object's known 3D model at its detected pose. This isolates each known object in its own spatial region. Object detection is provided by an oracle that returns ground-truth object identities and poses from the simulator; an object counts as visible when at least one ray cast from the camera to a corner of its bounding box reaches it unobstructed. This isolates the planning contribution from perception noise; replacing the oracle with a learned, image-based detector is left to future work.
2. **Occlusion-Aware Subdivision:** For each object-bounding Boxel, the space occluded by it from the fixed camera viewpoint is calculated and partitioned into one or more shadow Boxels, split by any intervening visible obstacles. As the robot acts, the set of shadow Boxels is updated: sensing a region resolves its shadow, and discovering a previously hidden object adds a shadow for that object. A relocated object, however, is not given a new shadow at its destination. Re-creating shadows for moved objects could admit plans that never terminate, so it is deliberately omitted; handling it safely would require additional belief bookkeeping and is left to future work. This explicitly represents regions the robot currently cannot observe, so the planner can target them with information-gathering actions.
3. **Recursive Partitioning:** The space not occupied by object or occlusion Boxels is partitioned recursively, starting from a single Boxel that spans the whole workspace. A Boxel

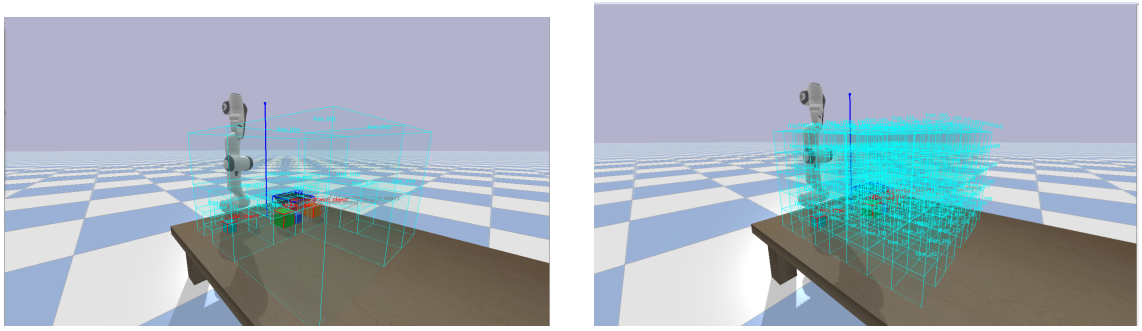
found to be empty is kept as a Free Space Boxel; a Boxel that still overlaps an object or occlusion is split into eight octants, and the procedure repeats on each. Subdivision stops when a region is empty or a minimum Boxel size is reached. The resulting free cells are then merged greedily across shared faces into larger convex Boxels, a pass repeated over multiple rounds until a round produces no further merges. This reduces their number without changing the represented free volume. This merge is convex-only (two cells are combined only when they share an exactly aligned face), so the partition retains more, smaller free Boxels than a full semantic merge would; this is accepted as sufficient for the object counts evaluated here (see Section 5.5).

Algorithm 1 Adaptive Semantic Discretization

Require: detected objects $\mathcal{O}$ (each with a pose and known 3D model), camera viewpoint c, workspace W, minimum leaf size $\ell_{\min}$
Ensure: Boxel set $\mathcal{BX}$

$\mathcal{BX} \leftarrow \varnothing$
for each object $o \in \mathcal{O}$ **do** *Step 1: object-centric bounding*
    add the tight axis-aligned box of o to $\mathcal{BX}$ as an `OBJECT` Boxel
for each `OBJECT` Boxel $b \in \mathcal{BX}$ **do** *Step 2: occlusion-aware subdivision*
    $s \leftarrow$ the region b occludes from c
    split s around any other object it sweeps over; add the fragments as `SHADOW` Boxels
$Q \leftarrow \{W\}$ *Step 3: recursive free-space partition*
while $Q \neq \varnothing$ **do**
    pop a cell q from Q
    if q overlaps no `OBJECT` or `SHADOW` Boxel **then**
        add q to $\mathcal{BX}$ as a `FREE_SPACE` Boxel *empty: kept*
    else if q is larger than $\ell_{\min}$ **then**
        push the eight octants of q onto Q *occupied: subdivide*
repeat *convex face-merge, to a fixed point*
    merge any two `FREE_SPACE` Boxels that share an exactly aligned face
until a round performs no merge
return $\mathcal{BX}$

This adaptive, object-centric approach contrasts with a uniform voxel grid by focusing discretization effort on task-relevant areas (the objects themselves and the spaces they hide from the camera viewpoint) rather than partitioning all space uniformly. Its free-space stage is itself an octree, and a uniform-grid baseline is implemented for comparison. Each generated Boxel is assigned a unique ID and associated geometric data; the completed set of Boxels is supplied to the planner as static facts in its initial state. Figure 4.3 contrasts the two partitions on a matching scene.



(a) Semantic partition: cyan Free Space Boxels where the scene is empty, plus object and shadow Boxels at the cube cluster.

(b) Uniform-grid baseline: hundreds of fixed-size cyan cells covering the whole workspace.

Figure 4.3: Adaptive semantic partition (left) versus uniform-grid baseline (right) on a matching scene. The adaptive partition produces a small set of task-relevant cells; the uniform grid covers the whole workspace at the same cell size, producing an order of magnitude more cells.

4.2 Belief over Boxels with Know-If Fluents

The Boxels generated in the previous section partition the workspace into a finite set of regions; what the robot does not know is which Boxel a hidden object occupies. We represent this uncertainty with the Know-If Fluents introduced in Chapter 2: one fluent per object and Boxel, marking whether the robot yet knows if the object is in that Boxel (known to be, known not to be, or unknown). Because the set of Boxels is finite, this representation is finite as well, and it is held directly in the planner's symbolic state. The Know-If fluents themselves are bookkeeping: they make what is known legible to a classical planner, while the actual work of resolving uncertainty is done by the sensing actions and the replanning loop that acts on their outcomes. Following Chapter 2, this is a Know-If fluent rather than a pair of K-literals; its realization in the domain is given in Listing 4.1.

Representing belief this way is what keeps planning efficient. As described in Chapter 2, carrying knowledge inside the state, whether as a pair of K-literals or in the lighter Know-If form we use (Section 4.2), compiles a partially observable problem into a classical one. In our domain this reduction lets an ordinary classical planner, under optimistic assumptions for sensing actions, reason directly about where objects might be and plan sensing actions to resolve that uncertainty within the same classical search, rather than delegating belief to a separate contingent or belief-space planner. The reduction itself is the one introduced by the POD planners CLG and LW1 [1, 6]; we adopt it in the weaker Know-If form rather than as full knowledge literals, and solve the result by optimistic determinisation with reactive replanning (Chapter 3), whose mechanics are given with the `sense` action below.

Knowing which Boxel holds an object is not enough on its own: a manipulation action also needs a reachable grasp and a collision-free trajectory. For this continuous-geometry side we rely on PDDLStream, which extends classical PDDL with *streams*, the lazily-evaluated samplers and tests for continuous parameters described in Chapter 2. Our approach combines the two in a single PDDL domain: Know-If Fluents carry the discrete belief over Boxels, while streams certify the continuous facts (grasps, inverse-kinematics solutions, and motions) that the manipulation actions require. A symbolic action such as picking a target object therefore mixes both kinds of precondition: a Know-If Fluent stating that the target's Boxel is known, and stream-certified geometric facts stating that the grasp and motion are feasible. The following two sections make this concrete, giving first the formal POD-TAMP model and then its PDDLStream realization.

4.3 POD-TAMP Model Definition

We formalize our POD-TAMP system as an explicit state model, adapting the state-space formulation of Geffner and Bonet [15] to the partially observable deterministic setting. The model is a tuple:

$$\mathcal{M} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{S}_G, \mathcal{A}, f, \mathcal{O} \rangle$$

where:

- $\mathcal{S}$ **(States):** A finite set of abstract states. Each state is a consistent assignment of truth values to a set of atomic propositions $\mathcal{AP}$ comprising the base facts and, over them, the Know-If Fluents introduced in Chapter 2. These propositions are grounded in the Boxel representation, e.g. the fact $\text{InBoxel}\big(obj_k, Boxel_j\big)$ and the Know-If fluent recording whether its value is known.
- $\mathcal{S}_0$ **(Initial States):** A set $\mathcal{S}_0 \subseteq \mathcal{S}$ of states consistent with the agent's initial knowledge, representing the initial belief b_0. Because a hidden target's Boxel is unknown, more than one initial state is possible, one for each Boxel the target could occupy; the belief is this set of candidate world states, and sensing shrinks it.
- $\mathcal{S}_G$ **(Goal States):** The set of states satisfying the goal formula over $\mathcal{AP}$ (e.g., K(holding(target_obj))). The task is solved only once the belief is contained in $\mathcal{S}_G$, i.e. the goal holds in every world still consistent with what the agent has observed.
- $\mathcal{A}$ **(Actions):** A set of abstract actions defined as PDDLStream action schemas that operate on the belief state, such as 'sense' or 'pick'.
- f **(Transition Function):** A deterministic state-transition function $s' = f(a, s)$ that applies an action's add and delete effects to the current state, so that the successor state reflects the updated knowledge.
- $\mathcal{O}$ **(Sensor Model):** The sensor model relates a sensing action to the observation it yields (the target is found in a Boxel, or it is not) and the resulting belief update. In our system

this is not realized as an observation function inside the POD planner: the planner's sensing action is optimistic and assumes the target is found, while the true outcome is obtained at execution time and folded into the belief by a Python sense-plan-act loop, which replans whenever the optimistic assumption proves wrong.

The planning problem is to find a sequence of actions from $\mathcal{A}$ that moves the belief state from $b_0 = \mathcal{S}_0$ to a belief state b_g in which every possible state is a goal state (i.e., $b_g \subseteq \mathcal{S}_G$). A sound POD solver for $\mathcal{M}$ would return a *contingent* plan that branches on each observation and drives the belief into $\mathcal{S}_G$ along every branch, reaching the goal with certainty for every world consistent with b_0; this is the guarantee provided by knowledge-literal POD planners such as CLG and LW1 [1, 6].

The system implemented in this thesis realises this model only approximately. Rather than solving the POD problem directly, it solves a single *optimistic determinisation* of $\mathcal{M}$: the `sense` action assumes its target is found, reducing the problem to one branch, which an ordinary classical planner solves. The plan is executed, and the true observation (supplied by $\mathcal{O}$, realised as a Python callable) triggers a replan whenever it contradicts the optimistic assumption (Section 4.2). The system therefore does *not* certify $b_g \subseteq \mathcal{S}_G$ over all states in the belief: the model above states the target semantics, including reaching the goal with certainty, while the determinise-and-replan realisation gives that up in exchange for staying within a single classical search. The soundness limits this incurs are consolidated in Section 5.5.

4.4 PDDLStream Integration

This section presents the implemented PDDL domain and streams that realize the model on the belief state over Boxels. The semantic discretization itself runs as a Python stage before each planner call, rather than as a PDDLStream procedure, so the Boxels enter each solve as fixed inputs. For readability the listings show the belief-related fragment of the domain, eliding the geometric and stacking predicates that are not central to this discussion.

4.4.1 PDDL Fluents for Belief Representation

The agent's knowledge is represented directly in the PDDL state. The implemented domain is untyped STRIPS, so object categories are themselves predicates, and the belief over object locations is carried by a single Know-If fluent.

```
1  (:predicates
2      ; --- Category predicates (untyped domain) ---
3      (Boxel ?x)              ; ?x is a Boxel
4      (Obj ?x)                ; ?x is a manipulable object
5
6      ; --- Boxel classification ---
7      (is_shadow ?b)          ; ?b is an occluded region
```

```
    (is_object ?b)           ; ?b bounds a physical object
    (is_free_space ?b)       ; ?b is known-empty space

    ; --- Object location: value and knowledge ---
    (obj_at_boxel ?o ?b)     ; object ?o is at Boxel ?b
    (obj_at_boxel_KIF ?o ?b) ; Know-If fluent: it is known whether ?o is at ?b

    ; --- Other knowledge and robot state ---
    (obj_pose_known ?o)      ; ?o has been localized to a Boxel
    (handempty)              ; the gripper is empty
    (holding ?o)             ; the gripper holds ?o
    ; ... further robot, geometry, and stacking predicates
)
```

Listing 4.1: PDDL fluents for belief over Boxels

A conceptual encoding would use the two K-literals $K(\text{InBoxel})$ and $K(\neg\,\text{InBoxel})$; the implemented domain instead uses the single Know-If fluent of Section 4.2: `obj_at_boxel_KIF` marks that the location of an object relative to a Boxel is *known*, while `obj_at_boxel` carries the value, and the absence of the Know-If fluent means the location is still uncertain. The two encodings are logically equivalent for this boolean case, and the single fluent keeps the grounded state smaller.

4.4.2 The Sensing Action

Information gathering is realized by a single `sense` action, which checks whether a target object is present in a given Boxel. A target counts as hidden, and so requires a sensing action, only when the camera's line of sight to every corner of its bounding box is blocked; the evaluation places each target to satisfy this.

```
(:action sense
    :parameters (?o ?region)
    :precondition (and
        (Obj ?o)
        (Boxel ?region)
        (view_clear ?region)                ; line of sight to the region is clear
        (not (obj_at_boxel_KIF ?o ?region)) ; only sense while the location is
            unknown
    )
    :effect (and
        (obj_at_boxel_KIF ?o ?region)       ; the location of ?o is now known
        (obj_at_boxel ?o ?region)           ; optimistic: assume the object is
            found here
        (obj_pose_known ?o)
        (increase (total-cost) 1)
    )
)
```

Listing 4.2: The implemented sense action

The action is *optimistic*: its effect unconditionally asserts that the object is found in the sensed region. An earlier design used conditional effects that branched on a `found`/`not_found` observation, which would let the planner build a multi-step search ("sense A; if empty, sense B") within one plan. PDDLStream and its classical backend do not support such contingent planning, so the implemented domain commits to the optimistic outcome and relies on *reactive replanning*: when execution reveals that the target is not in the sensed region, the execution loop updates the belief and replans. An empty region is recorded as clear; a region that instead turns out to hold a *non-target* object has that object (and the new region it occludes) added to the partition, re-boxelizing the workspace before the next plan so the newly revealed occluder enters the belief rather than being discarded. The loop continues until the target is located or the candidate regions are exhausted. Relocating a non-target object is a deliberate simplification: when the robot moves an occluder out of the way, our implementation does not recompute a fresh occlusion region behind its new pose, so moving an object cannot spawn new shadows to search. This optimistic determinise-and-replan scheme is therefore a deliberately *simplified* form of POD planning: a single optimistic plan repaired by replanning, rather than the branching contingent plans of a full POD planner; it suffices for the tasks evaluated here, but not for problems that genuinely require contingent reasoning (Section 5.5). Figure 4.4 shows the action targeting a shadow region in the simulator.

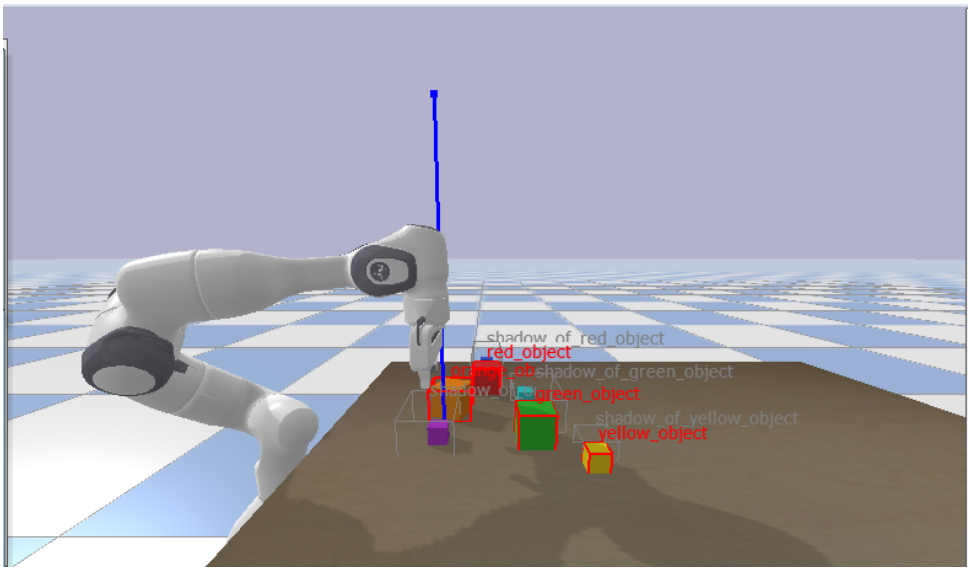


Figure 4.4: The `sense` action targeting a shadow region in PyBullet. The arm has been moved to a sensing configuration over a labelled shadow; the in-simulator overlay labels each object and its shadow region. This is the configuration in which the `sense` action's `(view_clear ?region)` precondition is checked and its optimistic `(obj_at_boxel ?o ?region)` effect is asserted.

This belief update is observation-driven: the reading from a `sense` action can only be obtained when `view_clear` holds for the region, so knowledge about a shadow is gated on first having a clear line of sight to it. This mirrors how sensing works in classic partially observable planning problems such as Wumpus World [40], where the agent smells a stench only at a square it actually occupies: the percept is not available on demand but rides on the action that establishes its precondition (moving onto the square there, clearing the view here) [1, 15]. A shadow whose view stays blocked therefore yields no observation and simply remains unknown; it cannot become known-empty by repeated attempts, since every attempt needs the same clear line of sight.

One bounded exception to this observation-driven belief update concerns regions that cannot be observed at all: if the line of sight to a shadow remains blocked after a fixed number of sensing attempts, that shadow is marked empty without ever being directly observed. This keeps the planner progressing when a region is effectively unreachable, but it is unsound: the region could still hold the target. A sound treatment, which would re-ground the view-blocking facts so the planner first clears the obstruction, is left to future work. The shortcut is needed because our Know-If encoding can only assume an outcome and repair it by replanning; a full POD planner with deductive axioms (e.g. LW1 [6]) could instead *derive* emptiness, through an axiom such as "a region is known empty once all of it has been observed clear": the conclusion would then follow from the accumulated observations rather than from an optimistic guess (Section 5.5).

The sense action calls no streams. Because a fixed camera observes the scene, no robot sensing configuration has to be sampled, and the only geometric precondition is the derived predicate `(view_clear ?region)`, which holds when no object blocks the line of sight to the region. Streams instead serve the manipulation actions: the domain declares `sample-grasp`, `plan-motion`, `compute-kin`, and `compute-stack-kin`, so that an action such as `pick` requires a valid grasp, an inverse-kinematics solution, and a collision-free motion, each certified by a stream.

4.4.3 Stream Implementations

The four streams are short Python procedures over the PyBullet world; their declarative PDDL specifications are listed in Section A.2. Since the planner treats them as black boxes, this section states what each one actually computes.

sample-grasp. Yields a single top-down grasp per object: the gripper points straight down, and the certified grasp places the end effector at a fixed 0.10 m clearance above the object's centre. Execution later lowers the arm from this approach pose to the contact pose with an inverse-kinematics solve seeded from the approach configuration, which keeps the arm in

the same solution branch the planner validated. Richer grasp sets (side grasps, orientation sampling) are deliberately out of scope (Section 5.5).

compute-kin. Computes an arm configuration that reaches the grasp pose over a given Boxel. It runs PyBullet's null-space inverse kinematics from several seed configurations (the rest pose plus fixed perturbations); each seed steers the iterative solver into a different local minimum, so the stream can yield several distinct arm configurations for the same pose. Solutions outside the joint limits are rejected. The yielded configuration records the target object's body id so that motion planning can exclude it from collision checks (at a pick pose, contact with the target is intended). Checking the configuration against the rest of the scene is deferred to `plan-motion`: earlier actions in the same plan may relocate objects, so a static planning-time collision check would wrongly reject configurations that are valid at execution time.

compute-stack-kin. The stacking variant of `compute-kin`: it reads the support object's bounding box and computes the end-effector height at which the held object's bottom face rests on the support's top face, then solves inverse kinematics as above.

plan-motion. Certifies a collision-free joint-space trajectory between two configurations. Both endpoints are checked for collisions first; the direct linear path is then tried, which covers most transit moves; if it collides, a bidirectional RRT-Connect search [29] explores the seven-dimensional joint space (at most 2000 iterations, 0.2 rad extension steps, 5 % goal bias, 8 collision samples per edge), and the found path is shortened by random shortcutting (75 attempts). All collision checks run against the full scene, minus the bodies the endpoint configurations mark as held or targeted. The stream yields a waypoint trajectory, or nothing if no path is found, in which case the planner abandons that action sequence.

Manipulation model. The manipulation actions run on a deliberately simple model, summarised here (Section 5.5 lists the full set of simplifications). Grasps are top-down at a single fixed height offset, and a picked object is attached to the gripper by a constraint while it is carried. A `place` or `stack` lowers the object and lets the simulator settle it under gravity; a small hardcoded lift afterwards, invisible to the planner, leaves the arm at a collision-free start for the next move. A run counts as a success only if a physics check confirms the goal physically holds after settling: a plan that reaches the goal symbolically while leaving the target not actually grasped is recorded as a *false success* and counted as a failure (Section 5.4.10), not a success. The success rates are therefore honest, but they are bounded by this manipulation model as much as by the planner.

4.5 The Sense–Plan–Act Loop

The Boxel partition, the Know-If belief (Section 4.2), and the PDDLStream domain above are driven by an outer sense–plan–act loop (Algorithm 2). Because the `sense` action is optimistic, a plan is valid only until an observation contradicts it, so the loop interleaves planning, execution, and belief update, replanning whenever reality diverges from the optimistic assumption. Boxelization (Algorithm 1) runs once at the start and again whenever the partition goes stale: after a placement consumes a free cell, or when sensing reveals a previously hidden object that needs its own shadow Boxel. The loop is time-bounded rather than replan-bounded: it ends when the goal holds, the planner returns no plan, every reachable region has been searched, or the per-cell wall-clock budget is exhausted.

Algorithm 2 Sense–Plan–Act Loop (reactive replanning)

Require: scene, goal g, wall-clock budget T
Ensure: success, or a give-up / timeout reason

```
detect visible objects; build 𝓑𝒳 with Algorithm 1
belief ← every shadow Boxel marked unknown
q ← home configuration
while g is not satisfied and the elapsed time is below T do
    if g is a search goal and no unknown shadow remains then
        return target-not-found   nothing left to sense; some regions may have stayed blocked
    if the partition is stale then   a placement consumed a cell, or a new occluder appeared
        re-boxelize the free space                          octree split + convex merge, diffed
    π ← plan for g from belief and q          optimistic: sense assumes the target is found
    if π is empty then
        return planner-failed
    for each action a in π do                                   execute until reality disagrees
        execute a in the world and update q
        if a is a sense action then
            observe the region by ray-casting from the fixed camera
            update belief      target found, region known-empty, or a new occluder revealed
            if the observation contradicts the optimistic assumption then
                break                                          abandon the rest of π and replan
return success
```

5 Results and Discussion

This chapter reports the experimental evaluation of the Boxel-based POD-TAMP framework. Section 5.1 describes the experimental setup as executed. Section 5.2 defines the measured quantities. Section 5.3 defines the variants compared, including the TAMPURA external reference. The results themselves follow in the remaining sections.

5.1 Experimental Setup

Simulator and robot. All experiments were run inside PyBullet [8], with a 7-DOF Franka Emika Panda as the manipulator on a tabletop.

Perception. Object detection is performed by an oracle that reads ground-truth object poses from the simulator and uses ray-casts from a fixed camera, positioned in front of and slightly above the table looking down at an oblique angle (Figure 5.1), to determine which objects, and which Boxels, are visible. The simulator also renders RGB-D images, but they are not processed inside the planning loop. The modelled uncertainty is occlusion uncertainty (which Boxel a hidden object occupies), resolved at run time by the `sense` action.

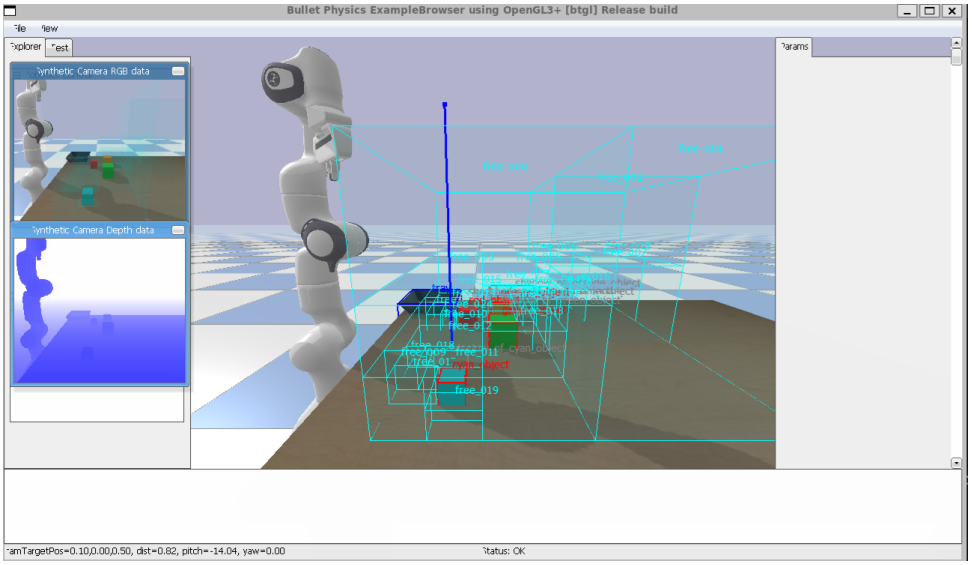
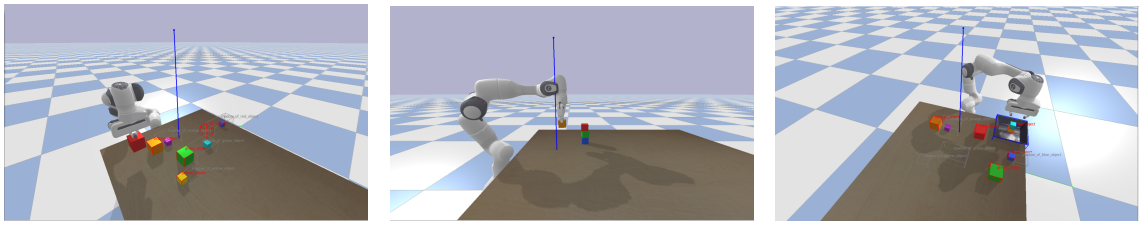


Figure 5.1: An evaluation scene observed by the fixed camera, shown here from an oblique simulator viewpoint; the corner panels are the camera's synthetic RGB (top) and depth (bottom) views. The cyan wireframe cells are the Free Space Boxels tiling the workspace; at the cube cluster, the red wireframe cells are the object Boxels bounding the occluders and the grey wireframe cells are the shadow Boxels marking the volumes they occlude from the camera. Object detection is an oracle that reads ground-truth object poses from the simulator rather than processing the rendered RGB-D images. A future learned detector would consume those images in place of the oracle, without changing the planning architecture.

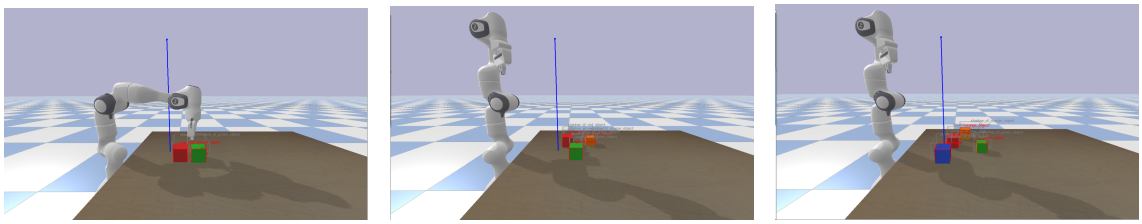
Goals. Three goals were evaluated (Figure 5.2):

- `holding` (*find*: locate a hidden target and pick it up): the robot must end the episode holding a single target object whose initial pose is hidden behind one of a set of occluders. The number of occluders n_{occ} is the difficulty axis (2, 3, 4; see Figure 5.3).
- `stack`: the robot must build a single tower of cubes to a given height h. The goal is a concrete conjunction of `(on` a b`)` facts (a specific tower) generated at random each episode; because the cubes are identical, which cubes form the tower is immaterial. The stack height h is the difficulty axis (2, 3, 4); cube positions are fully observable.
- `find-and-tray-stack` (*find and stack*): the robot must build a two-cube tower on a tray: a cube that starts in plain view on the table must be placed on the tray, and one hidden cube must first be located behind the occluders and then stacked on top; any further hidden cubes act as in-shadow distractors. The number of occluders n_{occ} is the difficulty axis (2, 3, 4).



(a) `holding` (b) `stack` (c) `find-and-tray-stack`

Figure 5.2: The three evaluated goals. `holding`: retrieve a single target hidden behind occluders. `stack`: build a tower of cubes to a target height. `find-and-tray-stack`: find a hidden cube and stack it on a tray.



(a) Low n_{occ} (b) Medium n_{occ} (c) High n_{occ}

Figure 5.3: The occluder-count difficulty axis n_{occ}, shown at low, medium, and high settings. More occluders crowd the target and occlude a larger share of the workspace, so the planner must reason over more shadow Boxels.

Scene generation. Scenes are generated procedurally from a seed, so the same seed always reproduces the same layout. A search-goal scene (`holding` or `find-and-tray-stack`) places n_{occ} occluder cubes (6–9 cm) at random reachable positions on the table, then hides K smaller cubes (4–5.6 cm) behind them, with K drawn uniformly from $[1, n_{\text{occ}}]$. Each small cube is set on an occluder's line of sight to the camera, so it cannot be seen until the robot senses that region; a post-spawn ray-cast verifies the occlusion, and the seed is re-rolled until every hidden cube is genuinely blocked; the re-roll sequence is itself derived deterministically from the original seed, so the same seed always yields the same final scene and the seed-for-seed pairing across variants is preserved. What each goal does with these cubes then differs: `holding` singles out one as the target to retrieve. `find-and-tray-stack` additionally spawns one small cube in plain view; the goal places that visible cube on the tray and stacks one hidden cube, once found, on top of it, while any remaining hidden cubes act only as in-shadow distractors. The `stack` goal involves no hiding, and spawns its cubes in plain view at random reachable positions.

Scenes and seeds. Each (goal, variant, difficulty) combination was run on 30 random scenes; seeds are paired across the `semantic` and `semantic+mbs0.05` variants (the latter forcing a 5 cm free-space leaf floor; Section 5.3) on the random-pairs scene, so their head-to-head comparison is seed-for-seed. The headline comparison spans 810 cells (3 goals × 3 variants × 3 difficulty

levels × 30 seeds). The free-space resolution-floor arms swept in Section 5.4.6 (floors from 1 mm to 18 cm) are reported separately there.

Planning budget. Each cell was given a wall-clock budget of 1800 s (30 min); a cell that exhausts it is recorded as *timed out*. This budget is a real operating constraint, not a rare safety net: 24 % of cells reach it (31 % on find-and-tray-stack, 24 % on stack, 16 % on holding). Individual planner calls are usually short (over successful cells the median call takes 1.3 s on stack, 3.4 s on holding, and 7.0 s on find-and-tray-stack), but the per-call distribution is heavy-tailed, with occasional calls running to hundreds of seconds. The long holding and find-and-tray-stack episodes therefore reflect this tail of expensive calls together with repeated sense–plan–act replanning, rather than many uniformly cheap calls; stack is the exception, with short episodes made up of many fast calls. The anytime sweep (Figure 5.4) records cumulative solve rate as a function of this budget.

5.2 Evaluation Metrics

For each cell we record:

- **Success**: whether the robot reached the goal within the planning budget.
- **Planning time**: total wall-clock time spent inside PDDLStream calls, summed across replans; this is the planning-time figure reported per goal in the headline results. It is distinct from the *end-to-end* per-episode wall-clock used in the TAMPURA comparison (Section 5.4.8), which additionally includes simulated execution and replanning: on holding that overhead adds about ten seconds (planning 134.30 s versus end-to-end 144.4 s), so the two are close but not identical.
- **Replan count**: how many times the planner was re-invoked in the sense-plan-act loop before terminating.
- **Representation size**: the Boxel set decomposed into `OBJECT`, `SHADOW`, and `FREE_SPACE` cells, and the number of grounded facts in PDDLStream's initial state. These two capture the per-cell cost of the partitioning strategy independently of planner choice.
- **Failure mode**: why a cell stopped without reaching the goal. We distinguish *timed out* (the planning budget was exhausted), *no plan found* (the planner returned no plan), *search exhausted* (the search ran out of candidate regions without ever observing the target: either every shadow Boxel was sensed and found empty while the target sat occluded just outside every shadow Boxel's bounds (an under-coverage of the axis-aligned shadow approximation), or the target's region stayed blocked from the fixed camera and the bounded give-up of Section 4.5 ended the search), and *false success* (the symbolic plan reached the goal, but a physics check found the world did not actually satisfy it (e.g. a stacked tower that had toppled, a grasp that did not hold, or a target displaced by contact during the final grasp approach), so the run is counted as a failure rather than a success).

A further give-up, a failed object release after repeated retries, occurred exactly once in the 810 cells (a find-and-tray-stack episode of the `semantic+mbs0.05` variant) and is counted as a failure.

Aggregate means in this chapter use one of two denominators. Metrics defined for every episode, such as the success rate, average over all n_{cells}. Metrics that exist only once the planner has recorded its first state (mean initial-state facts and mean replan count) average over the cells that have a value; about 18 % of cells lack one, having exhausted the planning budget before the first planner state was recorded, so these means are taken over the remaining cells.

5.3 Compared Variants

The headline comparison sets our method against a parameter-sensitivity check, an ablation baseline, and one external reference; beyond these, the resolution sweep of Section 5.4.6 adds further arms of the semantic variant, differing only in the free-space leaf-size floor (1 mm to 18 cm), which are defined and reported there.

1. **semantic** (our method): the adaptive Boxel partition described in Chapter 4. Free-space cells are placed only where the workspace is known to be empty; their leaf size is auto-set to the largest object's footprint (≈6–9 cm). This floor is a deliberate design choice rather than a tuning limit: each object is bounded by exactly one Boxel, so leaves finer than an object's footprint would fragment placement targets on the table dimensions used here.
2. **semantic+mbs0.05** (parameter-sensitivity check): the same adaptive partition but with the free-space leaf-size floor forced to 5 cm (`min_boxel_size = 0.05`). Because the auto-set leaf size is already coarser (≈6–9 cm) and the convex merge re-absorbs finer free cells, this floor never binds: the partition is identical to plain **semantic** (matching to ∼0.1 %). The variant therefore serves as a check that a finer free-space floor does not change the partition, not as a separate operating point.
3. **uniform** (ablation baseline): the adaptive partition is replaced by a uniform voxel grid over the workspace; the object and shadow Boxels are unchanged. The cell size is auto-set to the same largest-object-footprint value, since finer cells likewise break placement. This isolates the contribution of the adaptive free-space partition.
4. **TAMPURA** (external reference) [9]: a published POMDP-based TAMP framework. Its Table II reports several model-learning/search configurations on a *Partial Observability* task; we compare against TAMPURA's own primary configuration there, Bayes-Optimistic belief updates with LAO* search, which it reports at 57 ± 38 s over 20 trials (Table II of [9], arXiv v2), and we re-ran its released *Find Die* environment locally for a per-episode wall-clock; the comparison is architectural (a probabilistic learned-MDP planner solved with LAO* versus our deterministic knowledge-literal planning with replanning), not a hardware benchmark. See Section 5.4.9 for the full framing.

5.4 Results

Table 5.1 reports the aggregate success rate and mean success-only planning time for each (goal, variant) cell. The headline result is a capability gap: at the same cell scale, the adaptive partition lets the *same* planner solve substantially more scenes than the uniform grid, most sharply on the cluttered find-and-tray-stack goal (75.6 % vs 34.4 %) and by ≈1.4–2.2× in success rate across the three goals. The mechanism behind this is planning-time cost: because the adaptive partition grounds an order of magnitude fewer cells (Section 5.4.4), each planner call runs far faster, several-fold on holding (≈3.3×) and find-and-tray-stack (≈5.6×) and more than fiftyfold on stack (≈54×), so within a fixed planning budget more scenes finish in time. `semantic+mbs0.05` succeeds on exactly the same stack episodes as `semantic` (seed-for-seed identical), is modestly ahead on holding, and modestly behind on find-and-tray-stack. The subsections that interpret these results (Sections 5.4.5, 5.4.7, and 5.4.9) directly follow the results they discuss.

goal	variant	n_{cells}	success rate	mean plan time (s)
find-and-tray-stack	semantic	90	75.6 %	124.62
find-and-tray-stack	semantic+mbs0.05	90	72.2 %	82.14
find-and-tray-stack	uniform	90	34.4 %	692.11
holding	semantic	90	65.6 %	134.30
holding	semantic+mbs0.05	90	66.7 %	56.64
holding	uniform	90	47.8 %	438.53
stack	semantic	90	64.4 %	17.16
stack	semantic+mbs0.05	90	64.4 %	3.10
stack	uniform	90	35.6 %	924.13

Table 5.1: Headline success rate and mean planning time per (goal, variant). Planning time is success-only. Each cell aggregates 90 episodes (30 seeds × 3 difficulty levels).

5.4.1 Anytime Solve Rate

Figure 5.4 plots cumulative solve rate against wall-clock budget. The main finding is that the adaptive partition dominates at every budget: the `uniform` curve starts later and plateaus lower on every goal, so no choice of time budget closes the gap. On stack, `uniform` stays at zero until ∼800 s before reaching 36 %, and on find-and-tray-stack it climbs slowly to 34 %, consistent with its order-of-magnitude higher per-call cost. The second finding is that the budget is load-bearing on the search goals but not on stack: the semantic curves reach their ∼64 % stack plateau almost immediately (within ∼15 s, matching the 17 s mean planning time), whereas on holding and find-and-tray-stack they climb steeply through the first ∼100 s and keep creeping upward toward the 1800 s budget, tracking the ∼124–134 s mean success-only

planning times, so a non-negligible share of solves arrives late. Finally, the 5 cm leaf floor changes nothing here: on stack the `semantic` and `semantic+mbs0.05` curves overlap exactly.

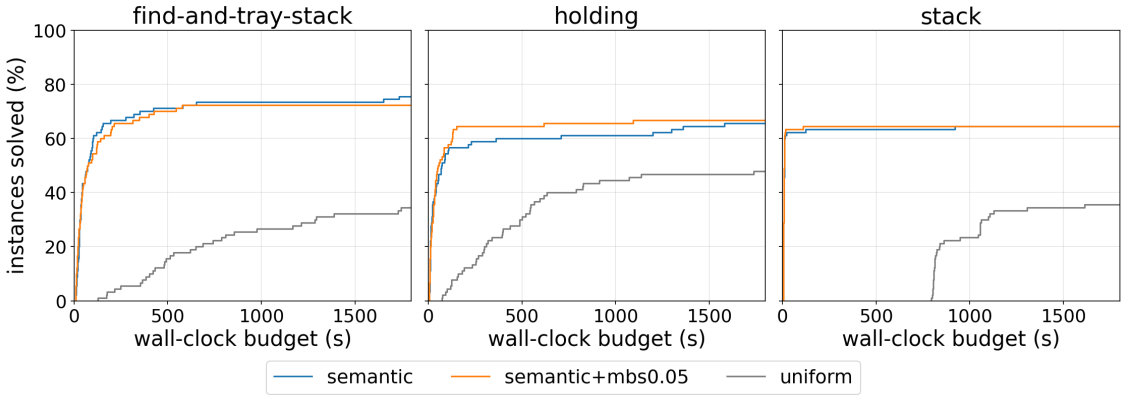


Figure 5.4: Cumulative solve rate vs wall-clock budget per (goal, variant). Semantic solves accrue within ∼15 s on stack but over hundreds of seconds on holding and find-and-tray-stack; `uniform` starts much later and plateaus lower on every goal. (On stack the two semantic curves coincide, so the orange overlaps the blue.)

5.4.2 Success Rate and Scene Difficulty

Figures 5.5 to 5.7 show success rate against the difficulty axis for each goal. On holding (Figure 5.5), success does not degrade with $n_{\text{occ}} \in \{2, 3, 4\}$; if anything it edges up at $n_{\text{occ}} = 4$: `semantic` runs 60–77 %, `semantic+mbs0.05` 63–70 %, `uniform` 43–53 %. There is no scalability collapse in this range, only a ∼18 pp gap between adaptive and uniform partitioning.

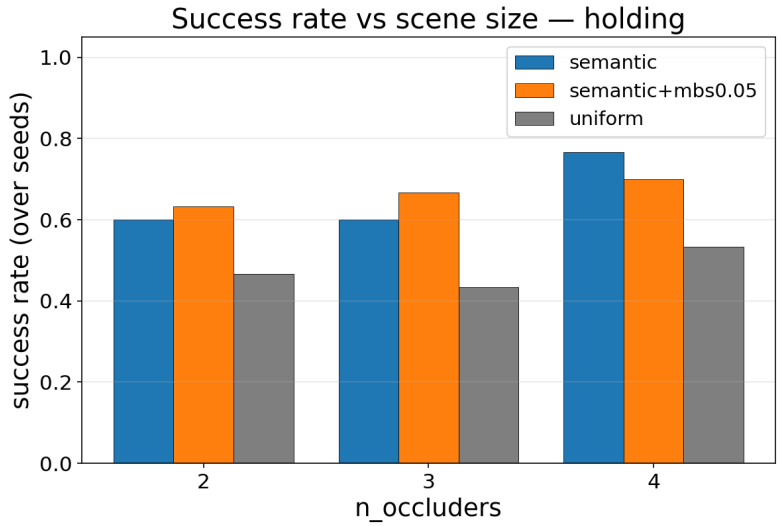


Figure 5.5: Success rate vs n_{occ} on the holding goal. Adaptive variants (semantic, semantic+mbs0.05) are within seed noise of each other; uniform sits ∼18 pp lower. Each bar is the mean success rate over the 30 seeds at that difficulty (here and in Figures 5.6 and 5.7).

On stack (Figure 5.6), the difficulty axis is the stack height. The adaptive variants solve $h = 2$ perfectly (100 %) and uniform nearly so (90 %); semantic and semantic+mbs0.05 then degrade gracefully to 80 % at $h = 3$ and 13.3 % at $h = 4$, while uniform collapses from 90 % to 16.7 % to 0 %. The semantic and semantic+mbs0.05 results are identical seed-for-seed: the same episodes succeed under both variants. This decline with height is not a planning effect: the stack goal is fully observable and the planner finds a valid plan to the target tower at every height. It reflects a limit of the simulated manipulation instead. As Section 5.5 details, execution uses a deliberately simplified manipulation model (top-down grasps at a single fixed height offset, a hardcoded post-place lift, and collisions that are logged but never trigger a recovery), so past a few layers the arm can no longer reliably place on or clear the growing tower, and upper-layer placements miss in execution even when the plan itself is sound. Because this ceiling is a property of the simulated arm and not of the partition, it lowers every variant as h grows; uniform falls off fastest, consistent with its order-of-magnitude higher per-call planning cost (Table 5.1). We read the stack-height limit as an accepted simplification of the manipulation stack rather than a failure of the partitioning or the planner.

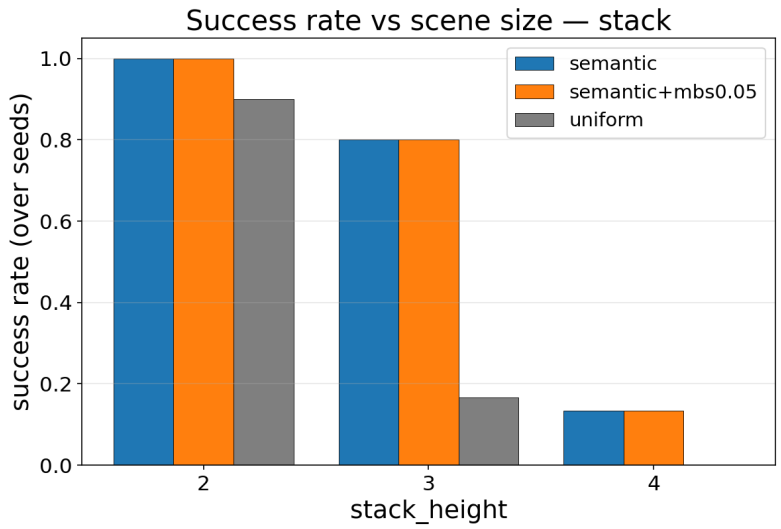


Figure 5.6: Success rate vs stack height h on the stack goal. Adaptive variants degrade gracefully; uniform collapses past $h = 2$.

Find-and-tray-stack (Figure 5.7) is the most discriminating goal: `uniform` trails at every difficulty (60.0 % / 16.7 % / 26.7 % at $n_{\text{occ}} = 2/3/4$), whereas `semantic` is far stronger and no longer flat, rising to 86.7 % at $n_{\text{occ}} = 4$ (73.3 % / 66.7 % / 86.7 %). The `semantic+mbs0.05` variant is non-monotonic here (70.0 % / 76.7 % / 70.0 %) and ends slightly behind `semantic`, the only goal where the finer leaf floor leaves a visible mark.

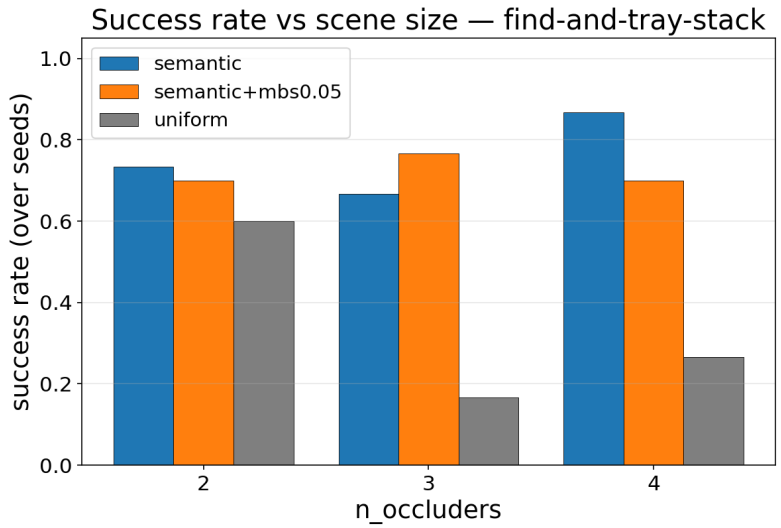


Figure 5.7: Success rate vs n_{occ} on the find-and-tray-stack goal. `uniform` trails the adaptive variants at every difficulty; `semantic+mbs0.05` is non-monotonic in n_{occ} and ends slightly behind `semantic`.

5.4.3 Planning Time and Scene Difficulty

Figure 5.8 shows mean success-only planning time on holding against n_{occ}. The adaptive variants stay far cheaper than uniform at every n_{occ}, though each successful episode absorbs many sense–plan–act replans: `semantic+mbs0.05` runs ∼38–71 s and `semantic` ∼85–234 s (peaking at $n_{\text{occ}} = 3$), while `uniform` climbs monotonically from 302 s at $n_{\text{occ}} = 2$ to 570 s at $n_{\text{occ}} = 4$, several times the adaptive cost. The slope is a representation effect (Section 5.4.4). Each added occluder casts another shadow region, and for every free cell on the camera's line of sight to that shadow the planner grounds a view-blocking fact (the facts that keep a later placement from re-blocking a cleared region), so the number of these facts scales with the free-cell count. The uniform grid carries an order of magnitude more free cells than the adaptive partition, so each extra occluder inflates its init state, and its planning time, far more steeply, which is why uniform climbs while the adaptive variants stay far cheaper. Planning times on stack and find-and-tray-stack show the same ordering (`semantic` 2–31 s on stack and 20–253 s on find-and-tray-stack; `uniform` 875–1189 s and 623–865 s respectively).

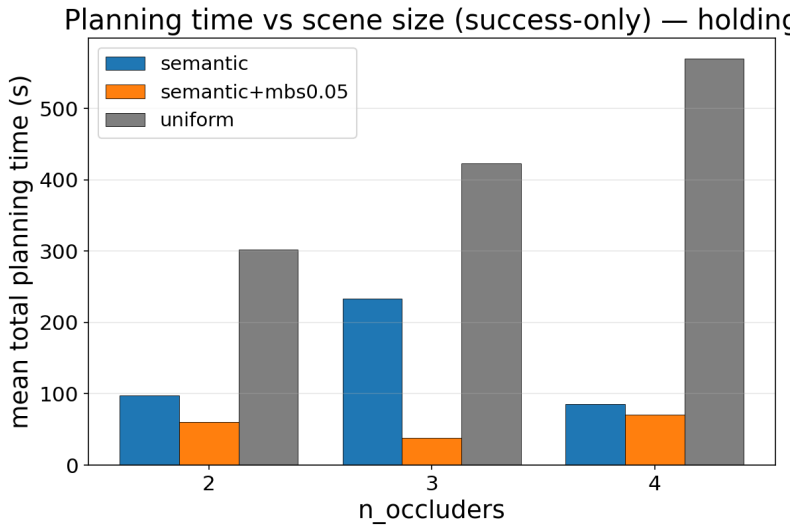


Figure 5.8: Mean success-only planning time vs n_{occ} on the holding goal. `uniform` stays several times more expensive and climbs steeply with n_{occ}, while the adaptive variants remain far cheaper.

5.4.4 Representation Compactness

The cost gap traces back to the size of the representation. Figure 5.9 stacks the OBJECT, SHADOW, and FREE_SPACE Boxel counts per variant on holding. The adaptive variants produce ∼22–38 Boxels total; the uniform variant produces ∼300 free-space cells alone. On the stack goal the ratio is steeper (semantic ∼25 vs uniform ∼1340), and the grounded-fact count scales with it: ∼300 facts for the adaptive variants vs 11 000–14 000 for uniform across stack heights. Larger init-state ⇒ larger PDDL grounding ⇒ longer per-call planning time, which is the mechanism behind the timing gap in Figure 5.8.

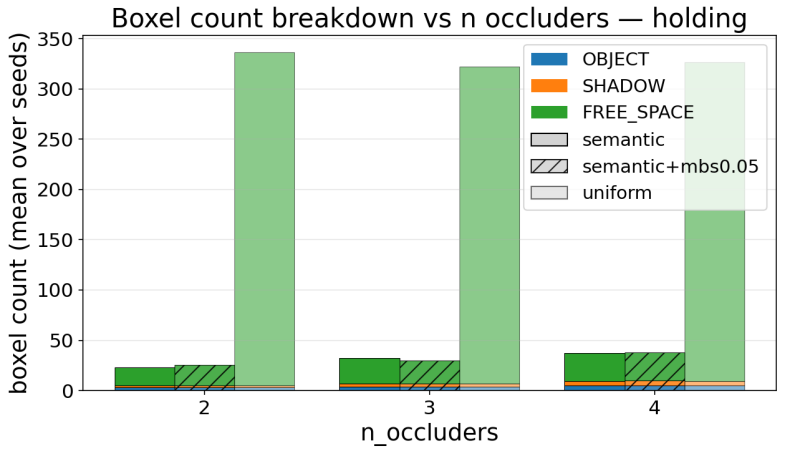


Figure 5.9: Mean Boxel count by category, holding goal. Adaptive variants produce an order of magnitude fewer cells than the uniform baseline.

5.4.5 Adaptive vs Uniform Partitioning

The headline finding is the cost gap between the adaptive partition and a uniform free-space grid at the same cell scale. Both partitions use the same auto-set cell size (the largest object's footprint, ≈6–9 cm in our scenes), so the comparison isolates the structural choice rather than the resolution. On holding, the adaptive partition produces 22–38 Boxels per scene; the uniform grid produces ∼300 free-space cells alone (Figure 5.9). Larger partitions translate to larger initial-state fact counts (∼300 vs 11 000–14 000 on stack) and longer per-call planning time, which is the mechanism behind Figure 5.8.

What that gap buys is goal-dependent. On holding the uniform baseline still solves about half of scenes; it pays several times the planning time at every n_{occ} but the mechanism still works. On stack the adaptive variants degrade gracefully from 100 % at $h = 2$ to 13 % at $h = 4$, while uniform collapses from 90 % to 0 %. On find-and-tray-stack, which composes occluded sensing with stacking, uniform is much weaker but still functional (34.4 % success, against `semantic`'s 75.6 %). Find-and-tray-stack also outperforms holding despite containing a search, because its terminal requirement differs: holding must end with a physically verified grasp of the target and its search can exhaust without an observation (the two holding-only failure modes of Section 5.4.10), whereas find-and-tray-stack ends at a height-two tower, the most reliable manipulation in the stack family. The reading is that the adaptive partition does not change the planner; it changes how many irrelevant cells the planner has to ground and search over. When the goal needs only a handful of candidate placements (a single hidden target), the overhead of a few hundred free-space cells is tolerable; when the goal requires composing placements across multiple replans, the overhead dominates. On the fully observable stack goal this grounding-cost effect is especially clean: with nothing hidden, the partition's occlusion structure is inactive and its free-space cells act only as placement candidates, so the ≈54× planning-time gap there reflects cell count alone, not occlusion reasoning. The same partition is reused across all goals for a single code path, rather than substituting a coarser placement grid when nothing is hidden.

5.4.6 Free-Space Resolution

The adaptive partition has one free parameter: the free-space leaf size. The `semantic` variant auto-sets it to the largest visible object's footprint plus a 1 cm margin (≈6–9 cm, scene-dependent: 9.2 cm on the random-pairs scenes, 6 cm on stack). To check whether this heuristic sits at a sensitive operating point, we swept leaf-size floors over two orders of magnitude around it, with the same 30 seeds per difficulty level as the headline arms: finer than auto at 1 mm, 1 cm, and 5 cm (the `semantic+mbs0.05` arm), and coarser than auto at 10 cm and at 1.5× and 2× the auto value (13.5 and 18 cm on the random-pairs goals; 9 and 12 cm on stack). The two finest floors are evaluated on the random-pairs goals at $n_{\text{occ}} \in \{2, 3\}$ (58–59 episodes per goal). Figure 5.10 plots the resulting Boxel count against the leaf size as grouped bars per goal.

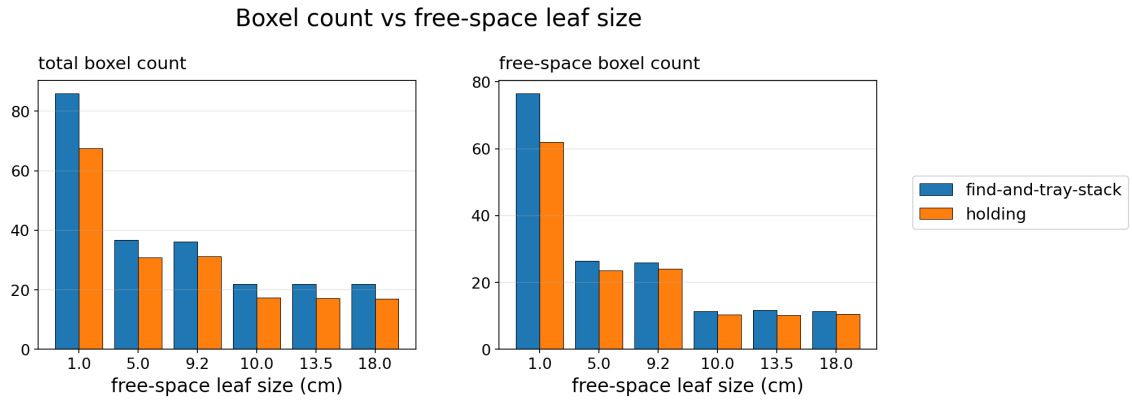


Figure 5.10: Mean Boxel count vs free-space leaf size for the find-and-tray-stack and holding goals (adaptive variants; total and free-space cells). Each bar group is one of six resolution arms (1, 5, 9.2 (auto), 10, 13.5, and 18 cm), labelled by its mean effective leaf size; the 1 cm bars average that arm's $n_{\text{occ}} \in \{2, 3\}$ episodes, and stack, which uses different floors, is reported in the text. The 1 mm arm is absent: none of its episodes completed within the planning budget, so it produced no partitions to count. The count is flat between 5 cm and auto, roughly triples at 1 cm, and the free-space count falls by ∼55 % as the leaf is coarsened past auto, flat from 10 to 18 cm.

Below auto the partition tolerates the floor until the floor gets extreme. Forcing the leaf floor to 5 cm leaves the partition unchanged (free-space counts within one cell of auto on every goal, and on stack identical seed-for-seed; Section 5.4.1), because the convex merge re-absorbs the finer leaves into the same regions. At a 1 cm floor the merge no longer fully re-absorbs the refinement: the free-space count is roughly three times the auto value (holding 62 vs 22 cells, find-and-tray-stack 77 vs 24, on the $n_{\text{occ}} \in \{2, 3\}$ scenes), yet success is unchanged on the same scenes (66.1 % vs 60.0 % on holding, 69.0 % vs 70.0 % on find-and-tray-stack). At a 1 mm floor the system stops producing results altogether: every episode (58–59 per goal) exhausted the 1800 s budget before completing. An over-fine floor therefore costs feasibility, not success.

Above auto the partition shrinks to a structural floor: coarsening to 10 cm collapses the free-space count by ∼55 % (holding 24.0 → 10.3, find-and-tray-stack 25.9 → 11.3), and the count then stays flat through 13.5 and 18 cm: the convex merge has already reduced free space to its convex structure, so past auto the leaf size stops mattering. Success moves little and with no trend in the leaf size: the coarse arms span 60.0–72.2 % on holding around auto's 65.6 %, and 62.2–73.3 % on find-and-tray-stack against auto's 75.6 %, while the smaller grounded problem plans faster on holding (mean success-only planning time 134 s at auto vs 34–47 s coarse; find-and-tray-stack is mixed, 77–132 s vs 125 s). On stack the resolution arms are seed-for-seed identical: all five adaptive settings (auto at 6 cm and floors at 5, 9, 10, and 12 cm) succeed on exactly the same 58 of 90 episodes; the 9 cm floor (1.5× auto) reproduces the auto partition outright, and the 10 and 12 cm floors shrink the free-space count by ∼32 % (17.6 → 11.9) with

no effect on the outcome. The leaf size is therefore a compactness-and-speed knob, not a success knob, except in the extreme: only the 1 mm floor breaks the system, and it does so through cost, not solution quality. The auto heuristic sits on a flat plateau rather than at a tuned optimum. Section 5.4.7 takes up why the partition is so insensitive.

5.4.7 Resolution Regime and the 5 cm Leaf-Floor Variant

The `semantic+mbs0.05` variant was included to test whether forcing a 5 cm floor on the free-space leaf size changes outcomes against the default `semantic` configuration, where the leaf floor is the largest object's footprint. It does not: on stack the curves overlap seed-for-seed; on holding the floor is ∼1 pp ahead of `semantic`; on find-and-tray-stack it is ∼3 pp behind, and non-monotonic in n_{occ} (70/77/70 %) rather than steadily degrading (Figure 5.7). On find-and-tray-stack the mean replan count rises relative to `semantic` (8.7 vs 7.6) without a matching gain in successes, consistent with the finer floor adding sensing/replanning cycles that do not pay off.

The mechanism behind the null finding, and behind the flat coarse arms of Section 5.4.6, is structural rather than numerical. The largest object's footprint (∼6–9 cm in our object set) is already the binding constraint on placement, and the merge step combines free-space cells whenever their merged region is convex, so the final partition reflects the convex structure of the free space rather than the leaf size that seeded it. Coarsening past *auto_cell* cannot remove that structure: the partition bottoms out at the workspace's convex decomposition, which is why the coarser floors all land on the same partition size. Refining below *auto_cell* gains nothing: the finer floor is either re-absorbed outright (the 5 cm null finding) or survives only as redundant subdivisions of the same free space (the 1 cm arm), inflating the grounding without changing what is reachable or searchable, so success stays put. The regime ends where this inflation outgrows the budget: at a 1 mm floor the partition and its grounding can no longer be processed in time, so the floor's failure mode is cost, not solution quality. *auto_cell* (the largest visible footprint plus a 1 cm margin) is therefore a convenient choice rather than a tuned constant, and semantic's compactness advantage over `uniform` is a structural property of the adaptive partition, not an artefact of resolution tuning. In scenes whose placements are constrained by sub-5 cm geometry (narrow pockets, dense small-occluder shelves), a finer leaf floor would in principle help; constructing such a scene and re-running the same comparison is the natural follow-up (Section 6.1).

5.4.8 Comparison with TAMPURA

Figure 5.11 compares per-episode times on our `holding` task, the closest analogue to TAMPURA's hidden-object *Find Die* task, against real TAMPURA, re-run on the same hardware. TAMPURA's published number for this task (57.0 ± 38 s, the *Partial Observability* column of Table II in [9]) is a *per-step* planning time that includes executing the selected controller in

simulation, not a per-episode figure; to obtain a like-for-like quantity we re-ran TAMPURA's released *Find Die* environment on this thesis's machine (20 seeds). Per successful episode it takes 166 ± 85 s, against our `semantic` variant's 144.4 s wall-clock (success-only, median 21.6 s); both are measured end-to-end, including simulated execution and replanning, on the same CPU, so the two are at rough parity per episode. Per step (wall-clock divided by the average number of plans), our cost is ∼11 s, against TAMPURA's published 57 s. On reliability, our `semantic` variant succeeds on 65.6 % of holding episodes, against the local TAMPURA run's 55 % and the ≥ 63 % implied by its published 0.63 ± 0.07 discounted return (Table I, $\gamma = 0.98$, binary terminal reward); the local re-run, with 20 seeds on hardware slower than the authors', may under-represent TAMPURA, and its 55 % is in any case within sampling noise of the published rate (the gap is under two episodes at $n = 20$), so the published-implied figure is the more conservative reference and is the one the abstract quotes. We are thus at rough parity on end-to-end cost and at least comparable in reliability, on a marginally easier goal (*Find Die* also requires returning home; our `holding` goal does not). The comparison is architectural, a probabilistic learned-MDP planner (LAO*) versus our deterministic knowledge-literal planning with replanning, and the two run on different environments, so it benchmarks neither system; Section 5.4.9 unpacks the framing.

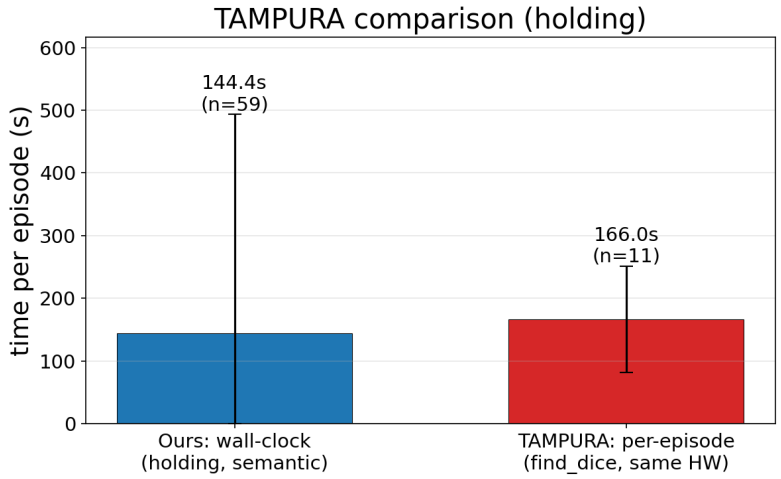


Figure 5.11: Per-episode time on `holding` (same hardware): our wall-clock (semantic variant; mean ± std, success-only; right-skewed, median 21.6 s) and real TAMPURA *Find Die* re-run locally (mean ± std over successful episodes; 20 seeds, 55 % success). Both are per-episode times including simulated execution. TAMPURA's published 57 s (Table II) is a per-step figure and is not plotted; see the text for the per-step reconciliation. The two are at rough parity per episode (144.4 s vs 166 s) and comparable in reliability: holding-`semantic` succeeds on 65.6 % of episodes against TAMPURA's local 55 % (its published returns imply ≥ 63 %).

5.4.9 Architectural Comparison with TAMPURA

The quantitative comparison is reported in Section 5.4.8 (Figure 5.11): against a local re-run of TAMPURA's *Find Die*, the closest analogue to our `holding` task, we are at rough parity end-to-end (144.4 s vs 166 s per episode) and comparable in reliability (65.6 % vs the ≥ 63 % implied by its published returns), on a marginally easier goal. Beyond the numbers, the two systems differ in kind.

Both systems sample geometry *online*, inside the planning loop, and both rely on the same determinised, Fast Downward–based classical planner, but they use it for different ends. TAMPURA's model-learning runs at every control step (Algorithm 2 of [9]). From the current belief it optimistically assumes each action succeeds and runs Fast Downward to propose candidate action sequences that reach the goal; it then samples those actions' real outcomes to estimate how they branch under uncertainty, building a *probabilistic* sparse MDP $\bar{\mathcal{B}}_{\text{sparse}}$, and finally solves that MDP for an uncertainty- and risk-aware policy. Fast Downward thus chooses *which goal-reaching action sequences are worth considering*; the MDP solver chooses *what to do once their outcomes are uncertain*, weighing each branch's probability of success. Our planner runs the same kind of determinised search through PDDLStream, but stops there: it reasons over deterministic knowledge literals, executes the chosen plan, and replans when an optimistic assumption proves wrong: there are no transition probabilities to estimate, no probabilistic MDP, and no policy layer on top. The salient difference is that probabilistic layer (learning the MDP and solving it for a policy), not the symbolic search engine underneath.

This framing changes what the time comparison means. It is not a "semantic-Boxels-beats-TAMPURA" claim. The two figures come from different environments (our tabletop scenes against TAMPURA's), so the times are not measuring the same problem and benchmark neither system. The planners do also differ in kind: ours runs deterministic classical search and replans, whereas TAMPURA learns a probabilistic sparse MDP and solves it for a policy. What the comparison does show is that a deterministic, replanning-based planner reaches the goal on the closest analogous task without ever estimating the learned sparse-MDP abstraction TAMPURA relies on. Both planners represent the belief with symbolic atoms (including epistemic ones, such as whether an object's pose is yet known), so the contrast is not whether knowledge is symbolic but how it is used: we reason over it deterministically and recover by replanning, with no transition probabilities to estimate, at the price of no risk-weighting over uncertain outcomes (Section 5.5).

Beyond *how* each system plans, the comparison highlights *what* each system can reason about. Because our planner holds occlusion as explicit symbolic state, line of sight is a planning condition: the `sense` action requires a clear view of its target region, and the objects that block a view are explicit facts in the planner's state. Those facts also cover candidate placements: a free cell lying on the camera's line of sight to a shadow is flagged as view-blocking, so the

planner will not set an object down where it would block a region it still needs to observe. TAMPURA's *Find Die* benchmark (its hidden-object task, the closest analogue to our setting) instead keeps occlusion sub-symbolic: a visibility voxel grid that only weights the success probability of the `look` action, and a placement sampler that draws object poses with no visibility check, so an object can be placed view-blind. Making space plannable in this way is the point of the representation.

Overall, the comparison's conclusion is that a deterministic knowledge-literal planner with replanning is sufficient to reach the goal on this class of partially observable tabletop tasks (at comparable end-to-end cost and reliability, and simpler to model than TAMPURA's learned probabilistic policy), and that it can plan over occlusions which a sub-symbolic visibility representation leaves implicit.

5.4.10 Failure Modes

Figure 5.12 stacks the exit reasons across the 6 (goal, variant) cells. *timed out* is the most common failure overall, dominating on find-and-tray-stack (`uniform` most of all) and on `stack semantic`, consistent with the lower success rates there rather than an inability to plan. *no plan found* is the main failure on `stack uniform`. The two holding-specific modes, *false success* (the run reached the goal symbolically but failed a physics check, e.g. the grasp did not actually hold) and *search exhausted* (the search ran out of candidate regions without ever observing the target), appear only on `holding`, where the target must actually be grasped and the search can end without the target ever being seen. The search-exhausted cells have two distinct causes. In 13 of the 18 such cells (across the three variants), every shadow Boxel was sensed and found empty while the target sat occluded just outside every shadow Boxel's bounds: the axis-aligned shadow volumes under-cover the occlusion cone near its edges, so a target at the cone boundary hides in a volume the partition never marks as unknown. In the remaining 5, the target's region stayed blocked from the fixed camera, and the bounded give-up (Section 4.5) ended the search without observing it.

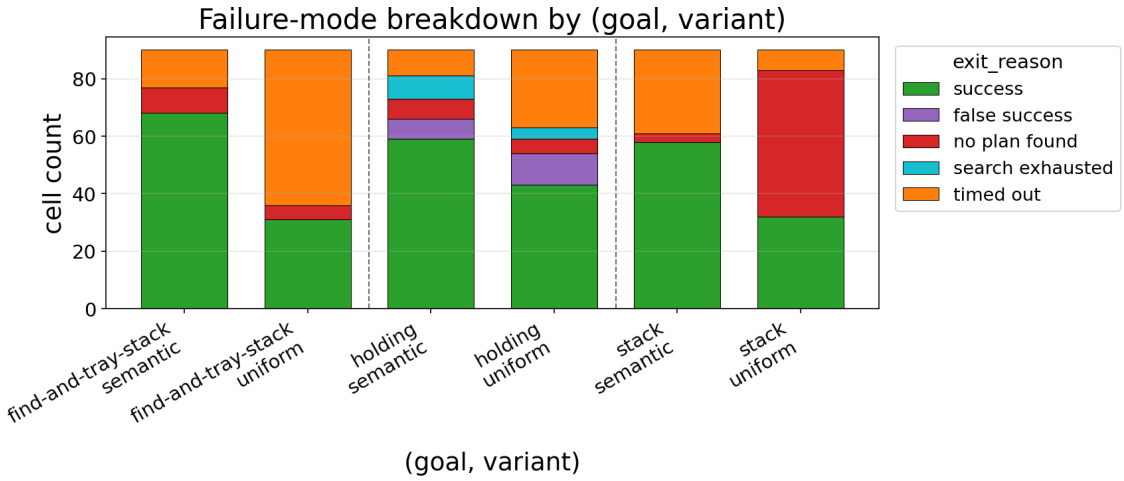


Figure 5.12: Stacked exit-reason breakdown per (goal, variant) for the `semantic` and `uniform` variants (90 cells each). Beyond *success*, a cell can end in *timed out* (the planning budget ran out), *no plan found* (the planner returned nothing), *search exhausted* (the search ran out of candidate regions without ever observing the target; Section 5.4.10 gives the two causes), or *false success* (the symbolic goal was reached but a physics check found it not actually satisfied, so it counts as a failure). *timed out* is the largest failure mode overall; *no plan found* dominates `stack` `uniform`; the holding-only modes (*search exhausted*, *false success*) reflect that `holding` requires actually grasping the target. The legend's *exit_reason* groups these outcomes.

The simplifications behind these numbers, and the scope they place on the claims, are consolidated in Section 5.5.

5.5 Limitations

The system evaluated in this thesis embodies a set of deliberate simplifications, and its findings are conditional on the scenes, the budget, and the oracle perception. Each simplification was adopted to keep the implementation tractable and to isolate the contribution (Boxel discretization and its integration with partially observable deterministic TAMP) from concerns that are separate from it. This section consolidates these simplifications and states the scope they place on the chapter's empirical claims.

Scene family. All measurements are on tabletop scenes with a fixed camera, a Franka Panda, and 6–9 cm cubes. The adaptive-vs-uniform gap is most pronounced in this regime because free-space cells dominate the uniform partition; in scenes with denser objects or smaller workspaces the ratio could narrow.

TAMPURA on a different task. We re-ran TAMPURA's released *Find Die* environment on this thesis's machine and cross-checked it against their published Table II, but we did not run

TAMPURA on our own tabletop scenes: *Find Die* is the closest analogue, not the same problem, so task, algorithmic, implementation, and parameter differences are bundled into a single delta. The scene diversity also differs: the released *Find Die* dataset ships three fixed scene layouts, one of which is drawn at random per episode, whereas our scenes are procedurally generated anew for every seed. Both planners are single-threaded. The per-episode figures we compare (144.4 vs 166 s, Section 5.4.8) were produced on the same machine, so they are not confounded by hardware. TAMPURA's *published* per-step time (57 s) was instead measured on the authors' Intel Xeon Gold 6248 (2.5 GHz base), against our AMD Ryzen 7 PRO 7730U (2.0 GHz base); that base-clock gap, if anything, slightly favours TAMPURA. Re-running both systems on a shared scene file would strengthen the comparison.

Perception. The scene contains a single fixed camera that observes the table from one viewpoint. Its rendered RGB-D images are not processed, however: object detection is performed by an oracle that queries ground-truth object poses directly from the simulator and uses ray-casts to determine which objects are visible from the camera's viewpoint. There is therefore no learned detector and no sensor-noise model. This isolates the planning contribution from perception error: a real perception module consuming the camera images could replace the oracle without altering the planning architecture, but would surface noise, misclassification, and partial-detection failure modes that the current numbers do not reflect. Because the camera is fixed, the robot never moves to a sensing configuration; this is sufficient for the tabletop scenes studied here, where every shadow is visible from one viewpoint, but a robot-mounted sensor would be required for occlusions not visible from any fixed vantage point. Several fixed cameras would fit the same architecture: a volume is hidden only if no camera sees it, so adding viewpoints shrinks the shadow regions without changing the planner. Deployment on real cameras additionally requires replacing the pose oracle with an image-based object detector, since a physical camera supplies pixels, not poses.

Manipulation and execution. Grasping is constraint-based: a held object is rigidly welded to the gripper, so objects cannot slip or be dropped in transit. Only top-down grasps are sampled, at a single fixed height offset. After a place or stack, execution adds a small hardcoded vertical lift before returning; this lift is invisible to the planner (it produces no PDDL state change and moves only along the gravity axis) and exists solely to give the sampling-based motion planner a collision-free start configuration for the next move. Collision events detected during execution are logged but do not trigger an automatic replan.

Spatial representation. Free-space cells are merged only across exactly aligned shared faces; a full semantic merge (matching view-blocking, observability, and ordering) was not implemented, so the partition contains more Boxels than strictly necessary. When an occluder is relocated, its previously computed shadow geometry and the shadow-blocker map are not

recomputed at its new position. This rests on the assumption that the world is static apart from the objects the robot itself moves: nothing else shifts, so the only outdated geometry belongs to the relocated object, whose new pose the planner tracks explicitly and replans from. The axis-aligned shadow Boxels also under-cover the occlusion cone: a target can sit camera-occluded just outside every shadow Boxel's bounds, in which case the search senses every candidate region, finds them all empty, and ends without ever observing the target. This is the dominant cause of the *search exhausted* failures (13 of 18 cells; Section 5.4.10).

Planning. The primary goal studied is `holding`; `stack` and `find-and-tray-stack` are shipped extensions that broaden goal diversity. A planner-search bias concerns the cost model: the domain gives the `stack` action a cost of 2 while every other action costs 1. This is a hand-tuned bias rather than a claim about stacking: because the planner minimises total plan cost and cannot weigh one action's outcome likelihood against another's, the higher cost is what keeps it from substituting a `stack` as a cheap rescue when a `place` onto a free Boxel fails its motion check, forcing it to exhaust placement options first. It is a deterministic stand-in for the probabilistic action selection the POD layer does not perform; `stack` remains available whenever the goal requires it. More fundamentally, when a shadow returns *still-blocked* three times in succession the execution loop marks it as not containing the target and proceeds. The shadow is never directly observed, so for a physically unreachable shadow this violates the invariant that belief should reflect observation. It is accepted to keep the sense–plan–act loop bounded; the run report distinguishes shadows observed empty from shadows given up this way, and the evaluation counts the give-up case as a failure. The principled remedy (re-grounding the planner's view-blocking atoms from current physics) is left to future work (Section 6.1).

Parameters. Several geometric constants (grasp height offsets, approach and lift heights, the octree's minimum leaf size, the camera pose) are tuned for the specific table, robot, and object set used here and would not transfer unchanged to a different setup. Generalizing them, for example by deriving grasp parameters from object geometry, is separate from the core contribution and is noted as future work.

Simplifications disclosed in the approach. Two further simplifications are stated where they are introduced rather than here. The conceptual belief could use two K-literals, $K(p)$ and $K(\neg p)$; the implemented domain instead carries a single Know-If fluent (recording whether p's value is known) paired with a value-carrying fluent, an equivalent encoding for the boolean known/unknown scenarios studied (Section 4.2). The `sense` action is modeled optimistically (it assumes the target is found), with reactive replanning when execution reveals otherwise, because PDDLStream and FastDownward support neither contingent branching nor deductive axioms (Chapter 4). The planning layer is thus a deliberately *simplified* form of POD planning:

a full POD planner such as LW1 [6] produces branching contingent plans and carries deductive axioms that infer facts the base predicates leave implicit, whereas ours determinises optimistically and repairs by replanning. This suffices for the single-target retrieval and stacking tasks studied here, where progress never hinges on a deduction the planner cannot make locally; it would not suffice for problems that genuinely require contingent reasoning: the unsound bounded give-up above is one symptom, since ruling out an unreachable shadow needs exactly the kind of deduction (clear the obstruction first) that an axiom-bearing POD planner expresses directly.

6 Conclusion

This thesis presented a novel approach to Task and Motion Planning under partial observability by integrating a semantic spatial abstraction, termed Boxels, with a Partially Observable Deterministic planning framework. At the core of the methodology is adaptive semantic discretization: it partitions the workspace into Boxels, concentrating detail on the objects and occluded regions that matter for the task. By modeling belief states over these Boxels, our system lets a PDDLStream-based TAMP framework plan around uncertain object locations and occlusions.

The POD-TAMP model formalizes this integration, enabling the planner to generate and execute plans that interleave information-gathering and manipulation actions. The use of Know-If Fluents to represent knowledge within the PDDL state allows for a direct and conceptually simple integration with existing POD planners. As demonstrated, this enables the robot to reason about what it knows and where uncertainty lies, and to take actions to resolve that uncertainty in a targeted manner.

This work contributes a representation in which occlusion is an explicit symbolic state, letting a TAMP framework handle partial observability without enumerating an exhaustive state space and without the sub-symbolic visibility grid of a POMDP-based TAMP baseline. An ablation against a uniform grid at the same cell scale shows that the adaptive partition produces an order of magnitude fewer cells and solves the cluttered find-and-tray-stack scenes far more often than the uniform baseline, and an architectural comparison with the POMDP-based TAMPURA framework indicates that deterministic knowledge-literal planning with replanning reaches the goal at comparable end-to-end cost and reliability, without learning a probabilistic policy. The detailed figures behind these claims are reported in Chapter 5. The remaining directions for extending this work are outlined below.

6.1 Future Work

The simplifications consolidated in Section 5.5 each suggest a concrete next step. The following directions would extend the system beyond the tabletop scenes evaluated here without altering its core architecture.

Learned perception. The most direct extension is to replace the oracle detector with a learned perception module that operates on the rendered RGB-D images rather than ground-truth simulator poses. Because the oracle is confined behind a narrow detection interface, such

a module could be substituted without changing the Boxel discretization or the planning layer, turning the architecture's perception-agnosticism from a claim into a demonstrated property.

Active sensing with a robot-mounted camera. The current fixed camera observes the table from a single viewpoint. A robot-mounted sensor would require the planner to reason about *where to look*: a sensing-configuration stream that computes a robot pose from which a target region becomes visible, integrated as an additional precondition on the `sense` action. This is the prerequisite for scenes with occlusions not visible from any single fixed viewpoint.

Discovering objects beyond anticipated shadows. When the robot senses a shadow region, any object hidden there is discovered and added to the representation. The scenes studied here are arranged so that every place an object can hide is one such anticipated shadow region; the search-exhausted failures of Section 5.4.10 are the boundary cases this arrangement misses. Extending the approach to objects hidden inside concave geometry (such as a container or cupboard, whose interior is itself a hidden region that may hold further objects and occlusions) would require the partitioning to recurse: re-running over the newly revealed interior and recursively bounding whatever it exposes.

Full semantic free-space merge. Free-space cells are presently merged only across exactly aligned shared faces. A full semantic merge that also matches view-blocking, observability, and back-to-front ordering would yield a coarser, more task-relevant partition with fewer Boxels, reducing the planner's initial-state size.

Real-robot experiments. All results here are in simulation. The natural culmination of the extensions above is to run the system on a physical Franka Panda, the robot modelled in our scenes, combining the learned detector and the robot-mounted active-sensing camera on real hardware and exposing the representation to the perception noise, calibration error, and contact dynamics that the simulator and oracle abstract away. Because the planning layer is insulated from perception behind a narrow detection interface, the Boxel discretization and the POD planner should transfer unchanged; testing that transfer is the point of a real-robot study.

A PDDL Specification

This appendix lists the complete PDDL domain and stream definitions used by the PDDLStream planner throughout the thesis. The domain (Section A.1) declares the predicates, derived predicates, and the five actions `sense`, `move`, `pick`, `place`, and `stack`. The streams (Section A.2) declare the four samplers that PDDLStream invokes lazily during search: `sample-grasp`, `plan-motion`, `compute-kin`, and `compute-stack-kin`.

A.1 Domain

```
;; ============================================================================
;; Semantic Boxel TAMP Domain - PDDLStream Compatible (Untyped)
;; ============================================================================
;;
;; Models the robot's CAPABILITIES (sense, move, pick, place, stack) rather
;; than any specific scenario. Scenario-specific spatial relationships
;; (e.g., which objects block which regions) are expressed as problem-level
;; init facts using generic predicates like blocks_view_at.
;;
;; Uses derived predicates for visibility: blocks_view and view_clear are
;; computed automatically from object positions — actions only need to
;; update spatial state (obj_at_boxel), not view predicates.
;;
;; Uses Know-If fluents for partial observability.
;; Object types are encoded as predicates: (Boxel ?x), (Obj ?x), etc.

(define (domain boxel-tamp)
  (:requirements :strips :equality :derived-predicates :conditional-effects
      :action-costs)

  (:predicates
    ;; --- Type predicates ---
    (Boxel ?x)
    (Obj ?x)
    (Config ?x)
    (Trajectory ?x)
    (Grasp ?x)

    ;; --- Boxel classification ---
    (is_shadow ?b)           ; Region not directly visible from the camera
    (is_object ?b)           ; Physical object (can be picked, placed, stacked)
    (is_free_space ?b)       ; Known empty space
```

```
32
33      ;; --- Visibility geometry (static) ---
34      (blocks_view_at ?obj ?b ?region) ; When ?obj is at ?b, it blocks view to ?region
35
36      ;; --- Visibility state (derived — DO NOT use in action effects) ---
37      (blocks_view ?obj ?region) ; ?obj currently blocks view to ?region
38      (view_blocked ?region)     ; Some object blocks view to ?region
39      (view_clear ?region)       ; No object blocks view to ?region
40
41      ;; --- Ground truth (actual world state) ---
42      (obj_at_boxel ?o ?b)     ; Object ?o is physically at boxel ?b
43
44      ;; --- Know-If fluent (do we know the value?) ---
45      (obj_at_boxel_KIF ?o ?b) ; We know whether ?o is at ?b (true or false)
46
47      ;; --- Robot state ---
48      (at_config ?q)
49      (handempty)
50      (holding ?o)
51      (obj_pose_known ?o)
52
53      ;; --- Stream certified facts ---
54      (valid_grasp ?o ?g)           ; Grasp ?g valid for object ?o
55      (motion ?q1 ?q2 ?t)           ; Trajectory ?t from ?q1 to ?q2
56      (kin_solution ?o ?b ?g ?q)    ; Config ?q for picking ?o from ?b with ?g
57      (config_for_boxel ?q ?b)      ; Config ?q targets boxel ?b (EE inside ?b)
58      (boxel_fits ?o ?b)            ; Boxel ?b is large enough to contain ?o (place
            dest or sense region)
59      (on_surface ?b)               ; Boxel ?b rests on a support surface (table)
60
61      ;; --- Stacking ---
62      ;; (on ?o ?support) means ?o sits directly on top of ?support.
63      ;; (on_table ?o) is the explicit "?o rests on the table" counterpart.
64      ;; (clear ?o) means nothing is stacked on ?o.
65      (on ?o1 ?o2)
66      (on_table ?o)                 ; ?o sits directly on the table
67      (clear ?o)
68      (is_tray ?o)                  ; ?o is the fixed-base tray support
69      (stack_kin ?o ?on_obj ?g ?q)  ; IK config ?q to place ?o on top of ?on_obj
70    )
71
72    ;; ============================================================================
73    ;; ACTION COSTS: bias planner toward place over stack
74    ;; ============================================================================
75    ;; All actions cost 1 except stack (cost 2).  Without this, the planner
76    ;; treated stack and place as equally cheap and chose stack as a "rescue"
77    ;; whenever motion planning to a particular free boxel failed, instead of
78    ;; trying other free boxels.  Higher stack cost forces the search to
```

```
79    ;; exhaust place destinations before falling back to stack.
80    (:functions (total-cost))
81
82    ;; ============================================================================
83    ;; DERIVED PREDICATES: Visibility from object positions
84    ;; ============================================================================
85    ;; blocks_view_at is a static geometric fact in the init state.
86    ;; blocks_view is derived: true when an object is currently at a position
87    ;; that geometrically blocks a view corridor.
88    ;; view_clear is derived via stratified negation: true when no object
89    ;; blocks the view.
90
91    (:derived (blocks_view ?obj ?region)
92      (exists (?b)
93        (and (obj_at_boxel ?obj ?b)
94             (blocks_view_at ?obj ?b ?region))))
95
96    (:derived (view_blocked ?region)
97      (exists (?obj)
98        (blocks_view ?obj ?region)))
99
100   (:derived (view_clear ?region)
101     (and (Boxel ?region)
102          (is_shadow ?region)
103          (not (view_blocked ?region))))
104
105   ;; ============================================================================
106   ;; SENSE: Observe a region to check for an object
107   ;; ============================================================================
108   ;; Requires clear line of sight to the region.
109   ;; Uses the fixed scene camera — no robot positioning needed.
110   ;;
111   ;; Optimistic effect: assume the target is found.  If execution reveals
112   ;; otherwise, the outer loop marks the shadow empty and replans, which
113   ;; avoids contingent planning that PDDLStream and FastDownward do not
114   ;; support.
115   (:action sense
116     :parameters (?o ?region)
117     :precondition (and
118       (Obj ?o)
119       (Boxel ?region)
120       (view_clear ?region)
121       (not (obj_at_boxel_KIF ?o ?region))  ; Only sense if unknown
122     )
123     :effect (and
124       (obj_at_boxel_KIF ?o ?region)        ; Now we know
125       (obj_at_boxel ?o ?region)            ; OPTIMISTIC: assume found
126       (obj_pose_known ?o)
```

```
127       (increase (total-cost) 1)
128     )
129   )
130
131   ;; ============================================================================
132   ;; MOVE: Move robot from one configuration to another
133   ;; ============================================================================
134   ;; ?b is the destination boxel — the stream that produced ?q2 certifies
135   ;; that the end-effector at ?q2 is within boxel ?b.
136   (:action move
137     :parameters (?q1 ?q2 ?b ?t)
138     :precondition (and
139       (Config ?q1)
140       (Config ?q2)
141       (Boxel ?b)
142       (Trajectory ?t)
143       (at_config ?q1)
144       (config_for_boxel ?q2 ?b)
145       (motion ?q1 ?q2 ?t)
146     )
147     :effect (and
148       (at_config ?q2)
149       (not (at_config ?q1))
150       (increase (total-cost) 1)
151     )
152   )
153
154   ;; ============================================================================
155   ;; PICK: Pick up an object from a boxel
156   ;; ============================================================================
157   ;; Must KNOW object is there (KIF=true AND at=true).
158   (:action pick
159     :parameters (?o ?b ?g ?q)
160     :precondition (and
161       (Obj ?o)
162       (Boxel ?b)
163       (Grasp ?g)
164       (Config ?q)
165       (handempty)
166       (at_config ?q)
167       (clear ?o)                    ; can't pick an object with something stacked
              on top
168       (obj_at_boxel_KIF ?o ?b)      ; Must know
169       (obj_at_boxel ?o ?b)          ; Must be there
170       (kin_solution ?o ?b ?g ?q)
171     )
172     :effect (and
173       (holding ?o)
```

```
      (not (handempty))
      (not (obj_at_boxel ?o ?b))
      (not (on_table ?o))         ; picking lifts ?o off the table
                                  ; (idempotent if ?o was on a stack instead)
      ;; If ?o was stacked on ?x, picking ?o makes ?x clear again.  For
      ;; holding-goal runs (no (on ...) facts) this grounds to a no-op.
      (forall (?x)
        (when (on ?o ?x)
          (and (not (on ?o ?x)) (clear ?x))))
      (increase (total-cost) 1)
    )
  )

  ;; ============================================================================
  ;; PLACE: Place an object in a boxel
  ;; ============================================================================
  ;; Destination must be free space.
  (:action place
    :parameters (?o ?b ?g ?q)
    :precondition (and
      (Obj ?o)
      (Boxel ?b)
      (Grasp ?g)
      (Config ?q)
      (holding ?o)
      (at_config ?q)
      (is_free_space ?b)
      (on_surface ?b)
      (boxel_fits ?o ?b)
      (kin_solution ?o ?b ?g ?q)
    )
    :effect (and
      (handempty)
      (obj_at_boxel ?o ?b)
      (obj_at_boxel_KIF ?o ?b)
      (on_table ?o)               ; place lands ?o on the table
      (not (holding ?o))
      (not (is_free_space ?b))
      (increase (total-cost) 1)
    )
  )

  ;; ============================================================================
  ;; STACK: Place the held object on top of another object
  ;; ============================================================================
  ;; Mirrors place but the destination is an OBJECT boxel rather than a
  ;; FREE_SPACE boxel.  ``stack_kin`` is certified by compute-stack-kin,
  ;; which derives the EE target from ?on_obj's CURRENT top z + the held
```

```
222   ;; object's half-height; this is computed each time the planner is
223   ;; invoked so multi-step stacks tolerate per-step settling.
224   ;;
225   ;; (clear ?o) becomes true on the newly-stacked object so it is itself
226   ;; available as a support for a subsequent stack action.
227   (:action stack
228     :parameters (?o ?on_obj ?g ?q)
229     :precondition (and
230       (Obj ?o)
231       (Obj ?on_obj)
232       (Grasp ?g)
233       (Config ?q)
234       (holding ?o)
235       (at_config ?q)
236       (clear ?on_obj)
237       (stack_kin ?o ?on_obj ?g ?q)
238     )
239     :effect (and
240       (handempty)
241       (on ?o ?on_obj)
242       (clear ?o)
243       (obj_at_boxel ?o ?o)        ; OBJECT boxel ID equals object name (a stacked
              object's boxel is itself)
244       (obj_at_boxel_KIF ?o ?o)
245       (not (holding ?o))
246       (not (clear ?on_obj))
247       (increase (total-cost) 2)
248     )
249   )
250 )
```

Listing A.1: The Boxel TAMP domain.

A.2 Streams

```
1  (define (stream boxel-streams)
2
3    ;; Sample grasp poses for an object
4    (:stream sample-grasp
5      :inputs (?o)
6      :domain (Obj ?o)
7      :outputs (?g)
8      :certified (and (Grasp ?g) (valid_grasp ?o ?g))
9    )
10
11   ;; Plan motion between configurations
12   (:stream plan-motion
13     :inputs (?q1 ?q2)
```

```
14      :domain (and (Config ?q1) (Config ?q2))
15      :outputs (?t)
16      :certified (and (Trajectory ?t) (motion ?q1 ?q2 ?t))
17    )
18
19    ;; boxel_fits is NOT a stream --- the facts are emitted in the initial
20    ;; state.  As a test stream it forced adaptive search to re-evaluate
21    ;; every (object, free_boxel) pair across skeletons.
22
23    ;; Compute IK solution for picking
24    (:stream compute-kin
25      :inputs (?o ?b ?g)
26      :domain (and (Obj ?o) (Boxel ?b) (valid_grasp ?o ?g))
27      :outputs (?q)
28      :certified (and (Config ?q) (kin_solution ?o ?b ?g ?q) (config_for_boxel ?q ?b))
29    )
30
31    ;; Compute IK for stacking ?o on top of ?on_obj.
32    ;; Reads ?on_obj's CURRENT AABB top + held object half-height to derive
33    ;; the EE target.  Certifies config_for_boxel against ?on_obj so the
34    ;; preceding move action delivers the arm to the support's OBJECT boxel.
35    (:stream compute-stack-kin
36      :inputs (?o ?on_obj ?g)
37      :domain (and (Obj ?o) (Obj ?on_obj) (valid_grasp ?o ?g))
38      :outputs (?q)
39      :certified (and (Config ?q) (stack_kin ?o ?on_obj ?g ?q) (config_for_boxel ?q
            ?on_obj))
40    )
41  )
```

Listing A.2: Stream definitions.

List of Acronyms

AI	Artificial Intelligence
CR	Critical Region
FOND	Fully-Observable Non-Deterministic
IK	Inverse Kinematics
KIF	Know-If Fluent
LAO*	Loop AO* Search
LLM	Large Language Model
MDP	Markov Decision Process
PDDL	Planning Domain Definition Language
POMDP	Partially Observable Markov Decision Process
RBVD	Region-Based Voronoi Diagram
STRIPS	Stanford Research Institute Problem Solver
TAMP	Task and Motion Planning
TAMPURA	TAMP with Uncertainty and Risk Awareness
VLM	Vision-Language Model

List of Symbols

State-Space Planning Model

M	deterministic state-space model $\langle S, s_0, S_G, Act, A, f \rangle$
S	set of states
s	a state
s_0	initial state
s'	successor state of s
S_G	set of goal states
Act	set of all (ground) actions
$A(s)$	actions applicable in state s
a	an action
$f(a, s)$	deterministic state-transition function
σ	a plan, i.e. a sequence of actions
P	a PDDL problem

Belief and Knowledge

b	belief state (set of possible states)
p	a proposition (ground atom)
L	a literal
$K(\cdot)$	knowledge (K-literal) operator: its argument is known to hold
$\neg$	logical negation

POD-TAMP Model

$\mathcal{M}$	POD-TAMP model $\langle \mathcal{S}, \mathcal{S}_0, \mathcal{S}_G, \mathcal{A}, f, \mathcal{O} \rangle$
$\mathcal{S}$	set of abstract (belief) states
$\mathcal{S}_0$	set of initial abstract states
$\mathcal{S}_G$	set of goal states
$\mathcal{A}$	set of abstract action schemas
$\mathcal{O}$	sensor model
$\mathcal{AP}$	set of atomic propositions
b_0	initial belief
b_g	goal belief

Boxels

$\mathcal{BX}$	set of Boxels
obj	a manipulable object

Boxel	a Boxel (workspace region)
InBoxel	predicate: object obj is in *Boxel*
N	number of candidate regions to search

TAMPURA Comparison

$\bar{\mathcal{B}}_{\text{sparse}}$	probabilistic sparse MDP learned by TAMPURA

List of Figures

List of Tables

List of Algorithms

List of Listings

List of References

[1] A. Albore, H. Palacios, and H. Geffner, "A translation-based approach to contingent planning.," in *IJCAI*, vol. 9, 2009, pp. 1623–1628.

[2] Y. Bai, Z. Wang, Y. Liu, K. Luo, Y. Wen, M. Dai, W. Chen, Z. Chen, L. Liu, G. Li, and L. Lin, "Learning to see and act: Task-aware virtual view exploration for robotic manipulation," *arXiv preprint arXiv:2508.05186*, 2025.

[3] W. Bejjani, W. C. Agboh, M. R. Dogar, and M. Leonetti, "Occlusion-aware search for object retrieval in clutter," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2021, pp. 4678–4685.

[4] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso, "Planning in nondeterministic domains under partial observability via symbolic model checking," in *IJCAI*, 2001.

[5] B. Bonet and H. Geffner, "Planning under partial observability by classical replanning: Theory and experiments," in *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*, 2011, pp. 1936–1941.

[6] B. Bonet and H. Geffner, "Flexible and scalable partially observable planning with linear translations," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 28, 2014, pp. 2235–2241.

[7] M. Brenner and B. Nebel, "Continual planning and acting in dynamic multiagent environments," *Autonomous Agents and Multi-Agent Systems*, vol. 19, no. 3, pp. 297–331, 2009. DOI: 10.1007/s10458-009-9081-1

[8] E. Coumans and Y. Bai, *PyBullet, a Python module for physics simulation for games, robotics and machine learning*, http://pybullet.org, 2021.

[9] A. Curtis, G. Matheos, N. Gothoskar, V. Mansinghka, J. Tenenbaum, T. Lozano-Pérez, and L. P. Kaelbling, "Partially observable task and motion planning with uncertainty and risk awareness," *arXiv preprint arXiv:2403.10454v2*, 2024.

[10] A. Elfes, "Using occupancy grids for mobile robot perception and navigation," *Computer*, vol. 22, no. 6, pp. 46–57, 1989.

[11] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial intelligence*, vol. 2, no. 3-4, pp. 189–208, 1971.

[12] 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, no. 1, pp. 265–293, 2021.

[13] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, *PDDLStream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning*, arXiv:1802.08705. Also in Proc. of ICAPS 2020, 2018. arXiv: 1802.08705.

[14] 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 *IEEE International Conference on Robotics and Automation (ICRA)*, 2020, pp. 5678–5684.

[15] H. Geffner and B. Bonet, *A concise introduction to models and methods for automated planning*. Morgan & Claypool Publishers, 2013.

[16] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: theory and practice*. Morgan Kaufmann, 2004.

[17] N. Gothoskar, M. Ghavami, E. Li, A. Curtis, M. Noseworthy, K. Chung, B. Patton, W. T. Freeman, J. B. Tenenbaum, M. Klukas, and V. K. Mansinghka, *Bayes3D: Fast learning and inference in structured generative models of 3d objects and scenes*, arXiv:2312.08715, 2023. arXiv: 2312.08715.

[18] D. Hadfield-Menell, E. Groshev, R. Chitnis, and P. Abbeel, "Modular task and motion planning in belief space," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2015, pp. 4991–4998.

[19] E. A. Hansen and S. Zilberstein, "Lao*: A heuristic search algorithm that finds solutions with loops," *Artificial Intelligence*, vol. 129, no. 1-2, pp. 35–62, 2001.

[20] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

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

[22] J. Hoffmann and R. I. Brafman, "Contingent planning via heuristic forward search with implicit belief states," in *ICAPS*, 2005, pp. 71–80.

[23] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *Journal of Artificial Intelligence Research*, vol. 14, pp. 253–302, 2001.

[24] T. Hofmann and H. Geffner, "Learning generalized policies for fully observable non-deterministic planning domains," *arXiv preprint arXiv:2404.02499*, 2024.

[25] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "Octomap: An efficient probabilistic 3d mapping framework based on octrees," *Autonomous robots*, vol. 34, no. 3, pp. 189–206, 2013.

[26] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial intelligence*, vol. 101, no. 1-2, pp. 99–134, 1998.

[27] L. P. Kaelbling and T. Lozano-Pérez, "Integrated task and motion planning in belief space," *The International Journal of Robotics Research*, vol. 32, no. 9-10, pp. 1194–1227, 2013.

[28] Y. Kim, R. Arora, R. Martín-Martín, P. Stone, B. Abbatematteo, and Y. Sung, "Large-language-model-guided state estimation for partially observable task and motion planning," *arXiv preprint arXiv:2603.03704*, 2026.

[29] J. J. Kuffner and S. M. LaValle, "RRT-Connect: An efficient approach to single-query path planning," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2000, pp. 995–1001.

[30] Y. Ma, Y. Yuan, S. Wu, and H. Yuan, "A task and motion planning framework for partially observable household manipulation scenes," *Advanced Intelligent Systems*, vol. 7, no. 12, p. 2 400 897, 2025.

[31] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "PDDL – the planning domain definition language," AIPS-98 Planning Competition Committee, Tech. Rep. CVC TR-98-003, 1998.

[32] D. Molina, K. Kumar, and S. Srivastava, "Learn and link: Learning critical regions for efficient planning," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020, pp. 10 605–10 611.

[33] C. Muise, V. Belle, and S. A. McIlraith, "Computing contingent plans via fully observable non-deterministic planning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 28, 2014, pp. 2322–2329.

[34] R. K. Nayyar and S. Srivastava, "Autonomous option invention for continual hierarchical reinforcement learning and planning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, 2025, pp. 19 642–19 650.

[35] H. Palacios and H. Geffner, "Compiling uncertainty away: Solving conformant planning problems using a classical planner (sometimes)," in *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI)*, 2006, pp. 900–905.

[36] T. Pan, R. Shome, and L. E. Kavraki, "Task and motion planning for execution in the real," *IEEE Transactions on Robotics*, vol. 40, pp. 3356–3371, 2024.

[37] R. P. A. Petrick and F. Bacchus, "A knowledge-based approach to planning with incomplete information and sensing," in *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems (AIPS)*, 2002, pp. 212–222.

[38] S. Richter and M. Westphal, "The LAMA planner: Guiding cost-based anytime planning with landmarks," *Journal of Artificial Intelligence Research*, vol. 39, pp. 127–177, 2010.

[39] G. Riegler, A. Osman Ulusoy, and A. Geiger, "Octnet: Learning deep 3d representations at high resolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 6620–6629.

[40] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Pearson, 2010.

[41] M. S. Saleem, R. Veerapaneni, and M. Likhachev, "A pomdp-based hierarchical planning framework for manipulation under pose uncertainty," *arXiv preprint arXiv:2409.18775*, 2024.

[42] N. Shah and S. Srivastava, "Using deep learning to bootstrap abstractions for hierarchical robot planning," in *Proc. of the 21st International Conference on Autonomous Agents and Multiagent Systems (AAMAS '22)*, 2022, pp. 1183–1191.

[43] L. Zhao, W. McClinton, A. Curtis, N. Kumar, T. Silver, L. P. Kaelbling, and L. L. Wong, "Seeing is believing: Belief-space planning with foundation models as uncertainty estimators," *arXiv preprint arXiv:2504.03245*, 2025.