# Task and Motion Planning

Caelan Garrett

11/06/2018

6.881 - Intelligent Robot Manipulation

http://manipulation.csail.mit.edu/

https://github.com/caelan/pddlstream

# (Probable) Roadmap



1. **Review** motion planning & AI planning

2. **Introduce** task and motion planning

3. **Survey** existing approaches

4. **Dive deep** into our formalism - **PDDLStream**

5. **Present** several PDDLStream algorithms

6. **Describe** extensions

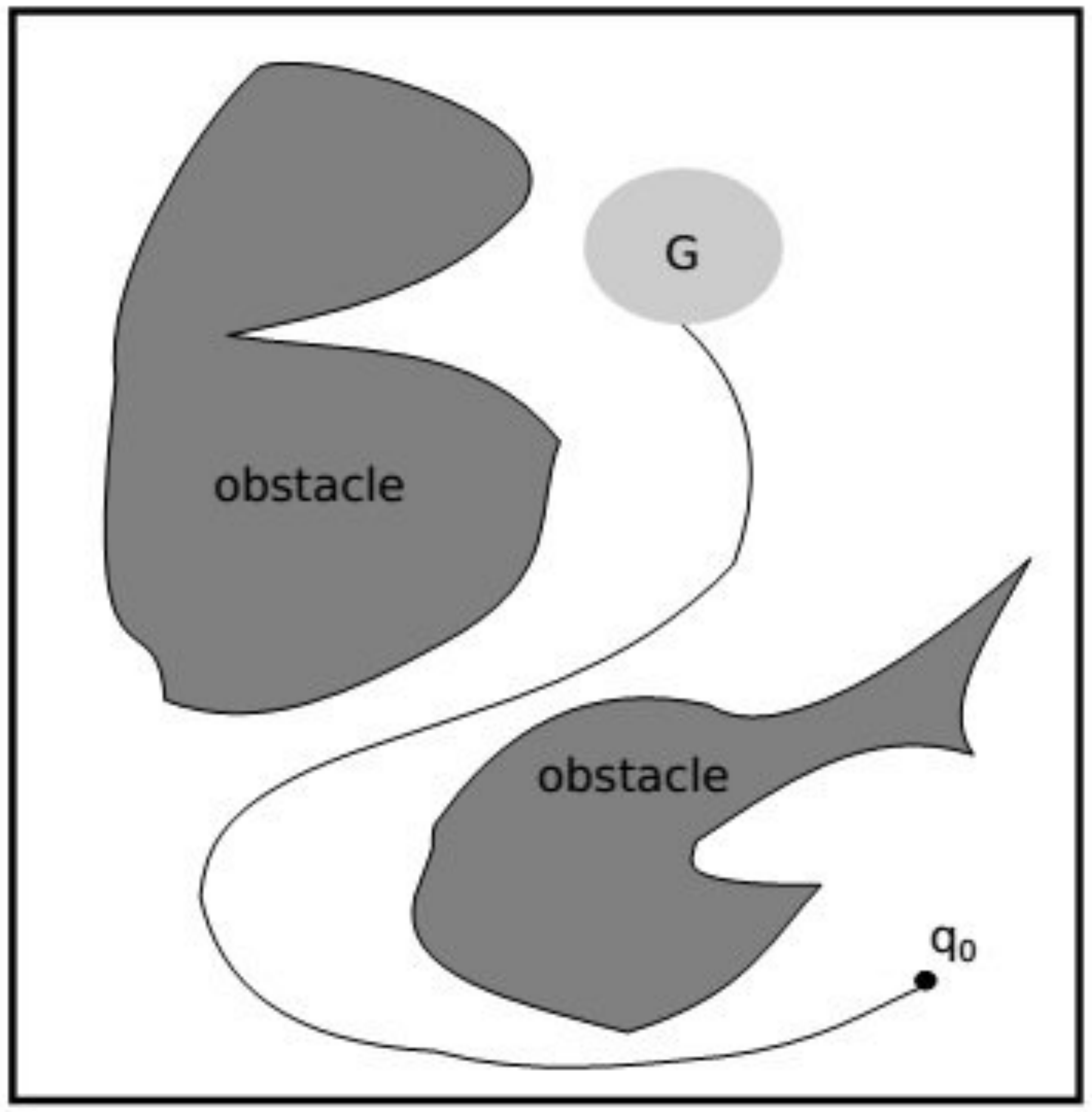
# Motion Planning (10/30/2018)

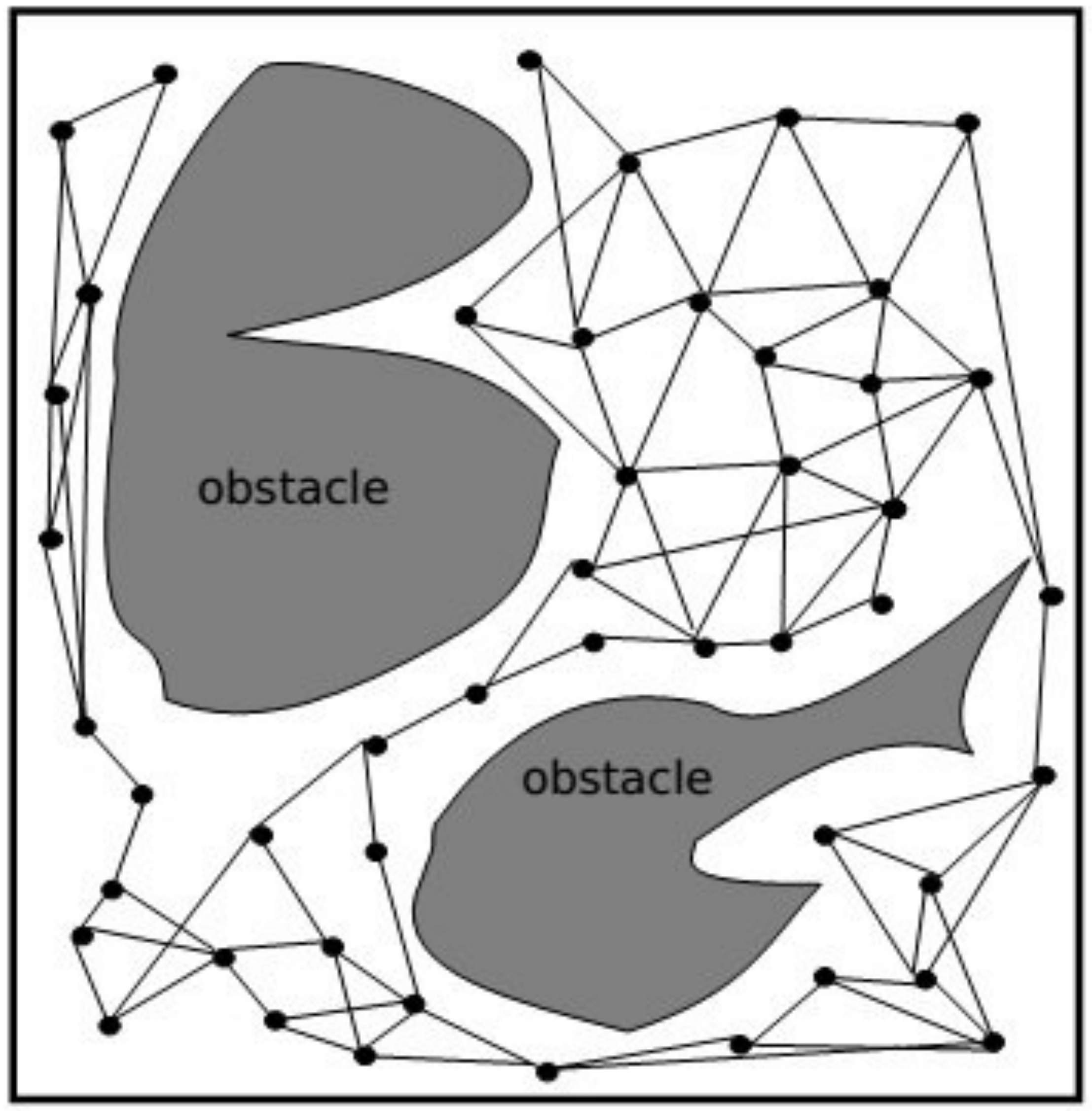


- Plan in a **continuous** configuration space
- **Sampling-based** motion planning

1. **Sample** robot configurations (randomly)
2. **Connect** nearby configurations if collision-free path
3. **Search** for a path within resulting graph

- PRM
- RRT
- RRT*



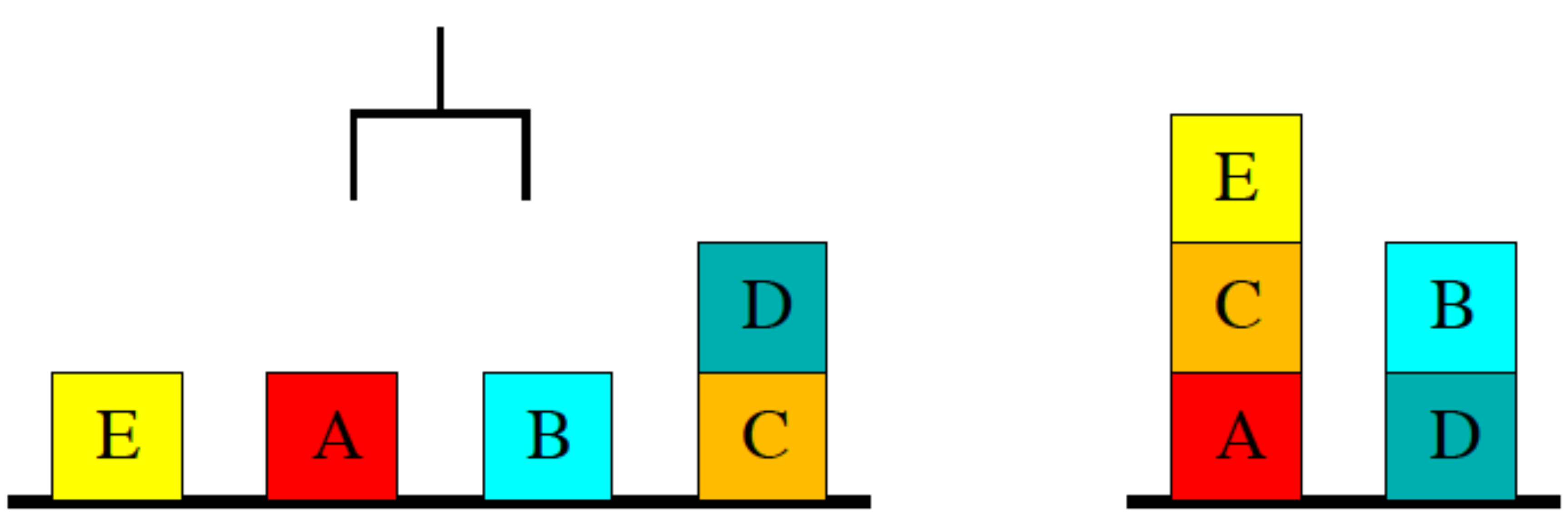



# AI (Task) Planning (11/01/2018)



- Plan in a large **discrete** space with **many variables**
- **Logical** descriptions - STRIPS/PDDL
  - Logical propositions = boolean variables
  - Parameterized actions
    - Preconditions & effects
- **Heuristic search** algorithms
  - Delete-relaxation ($h_{FF}$)

```
(define (domain blocksworld)
(:predicates (clear ?x) (holding ?x) (on ?x ?y)
             (on-table ?x) (arm-empty))
(:action stack
 :parameters (?x ?y)
 :precondition (and (clear ?y) (holding ?x))
 :effect (and (arm-empty) (on ?x ?y)
              (not (clear ?y)) (not (holding ?x)))
)
...
(define (problem bw-abcde)
(:domain blocksworld)
(:objects a b c d e)
(:init (on-table a) (clear a)
       (on-table b) (clear b)
       (on-table e) (clear e)
       (on-table c) (on d c) (clear d)
       (arm-empty))
(:goal (and (on e c) (on c a) (on b d))))
```




Initial State Goal State

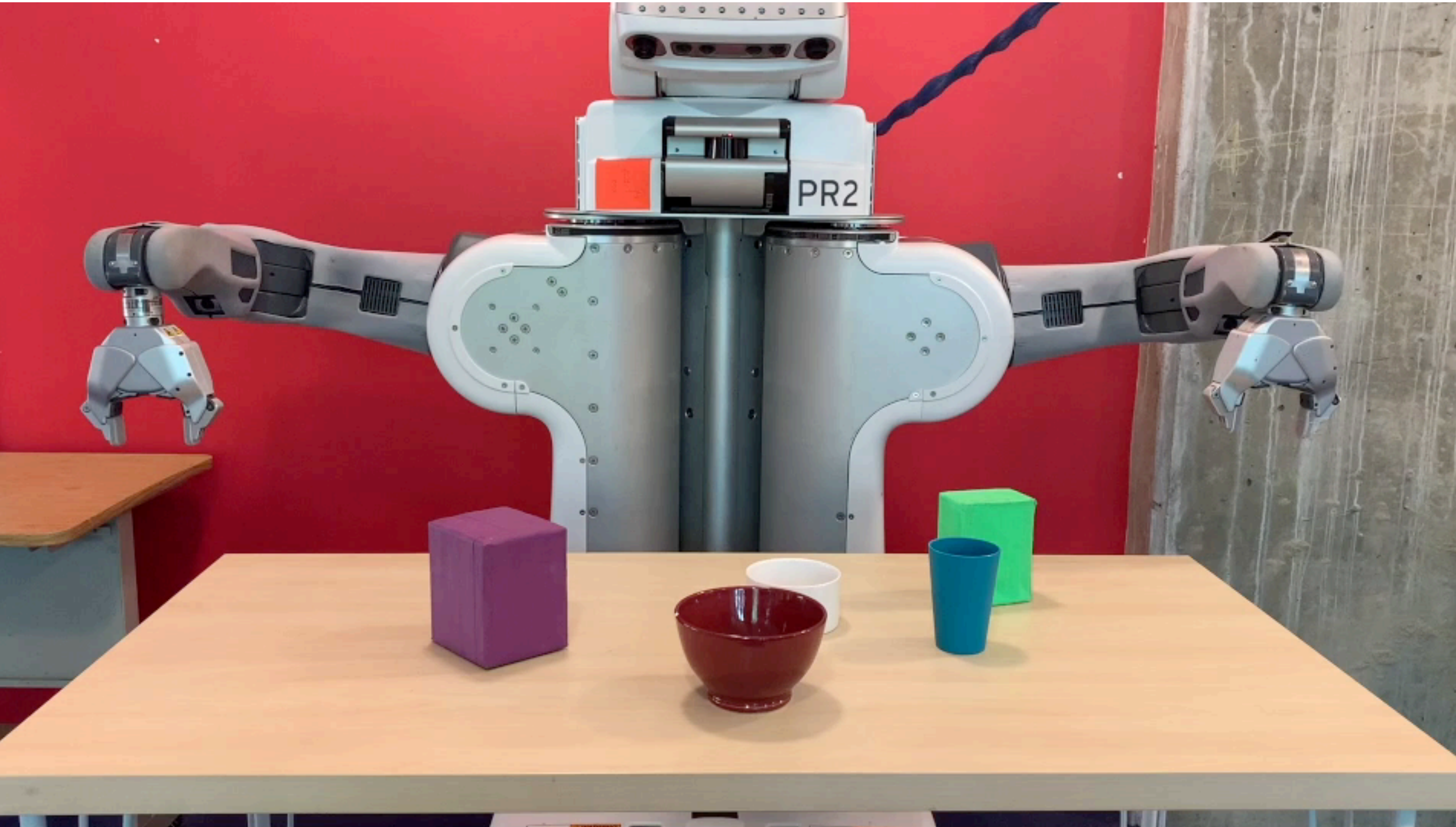
# Task and Motion Planning (TAMP)



- Plan in a **hybrid** space with many variables
  - **Discrete** and **continuous** variables & actions
- **Multi-step manipulation**
  - **Variables** - robot configuration, object poses, door joint positions, is-on, is-in-hand, is-holding-water, is-cooked, …
  - **Actions** - move, pick, place, push, pull, pour, cook, …

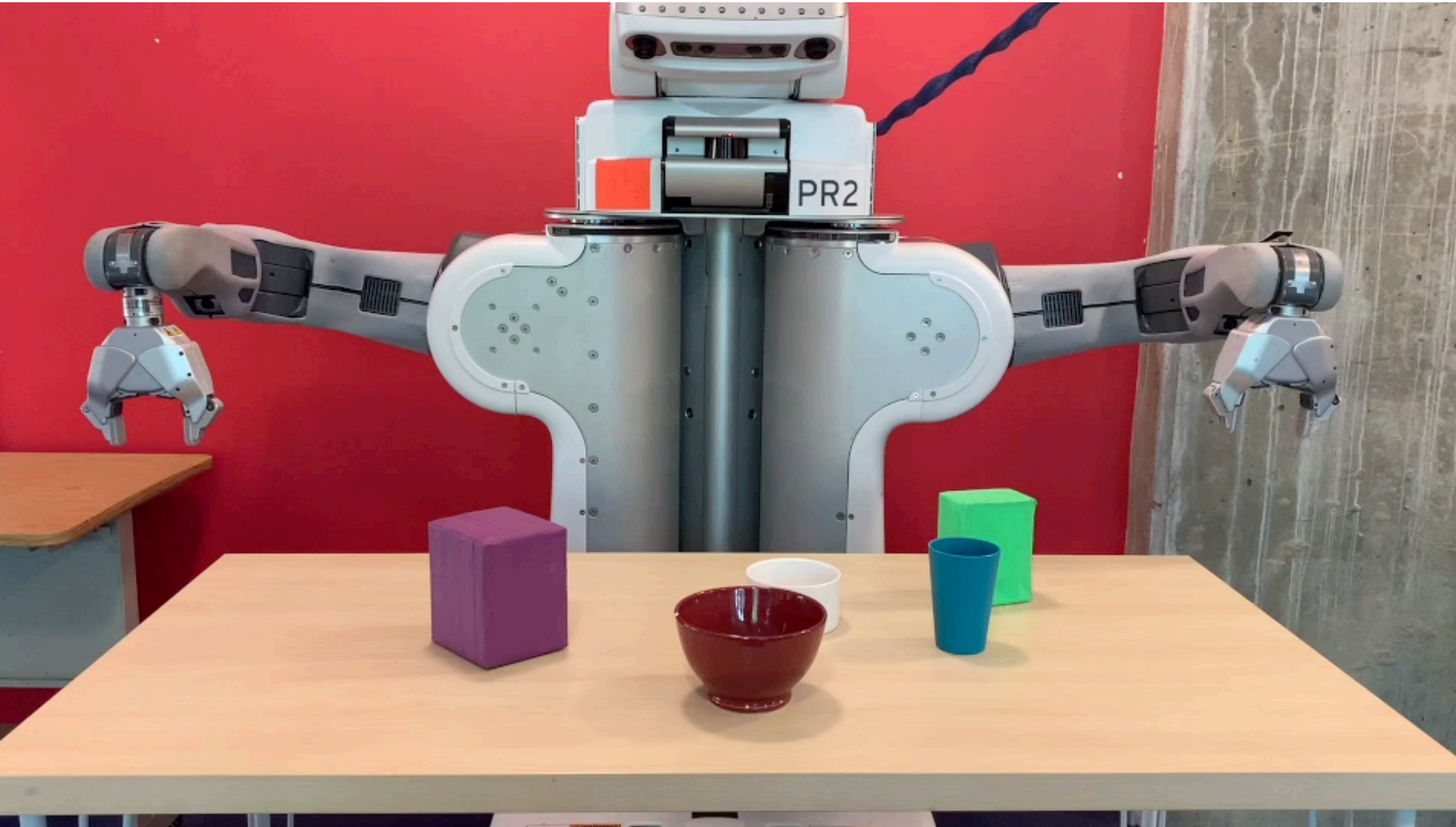
# PR2 Tabletop Manipulation





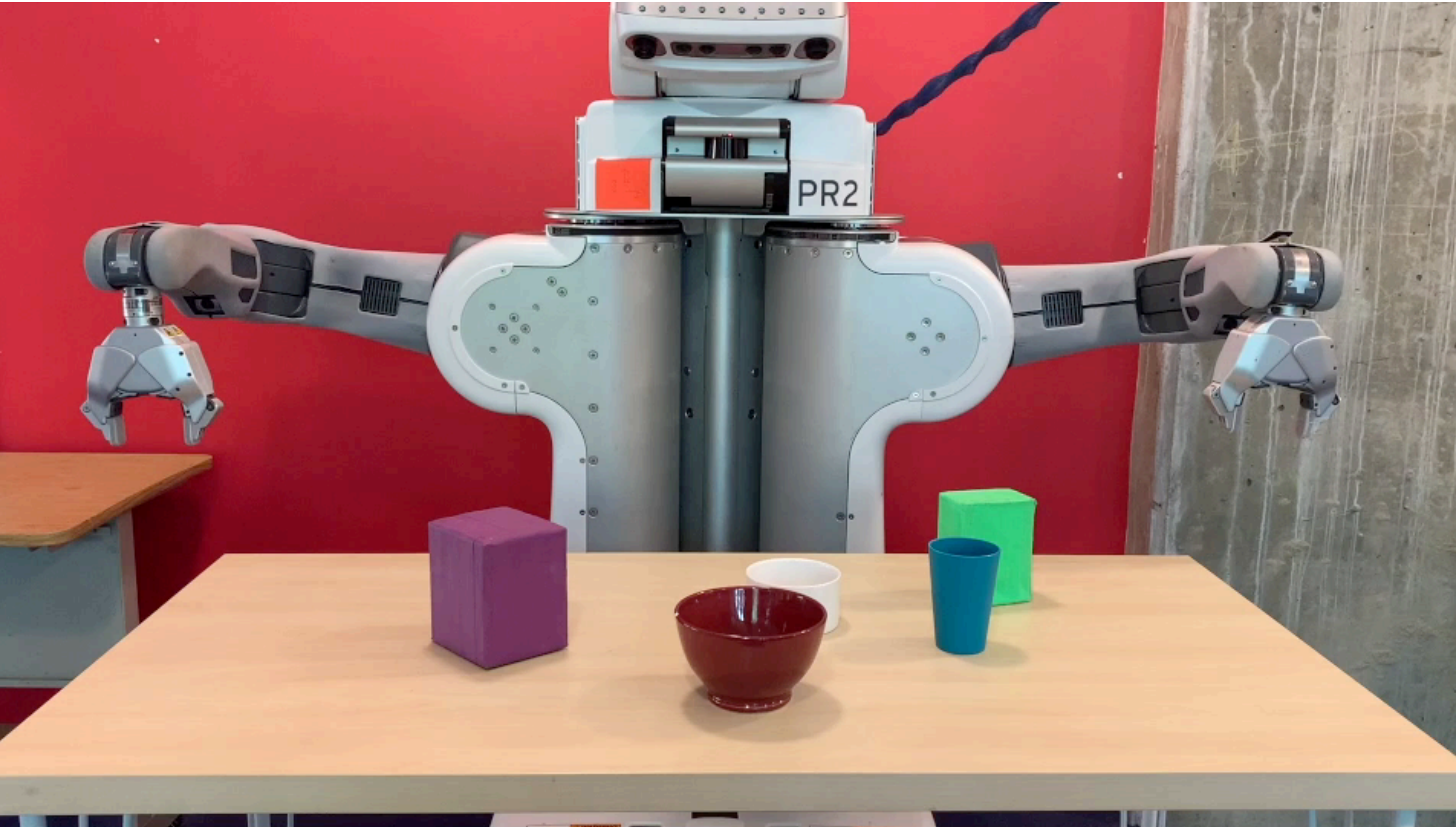
# PR2 Tabletop Manipulation





# PR2 Tabletop Manipulation

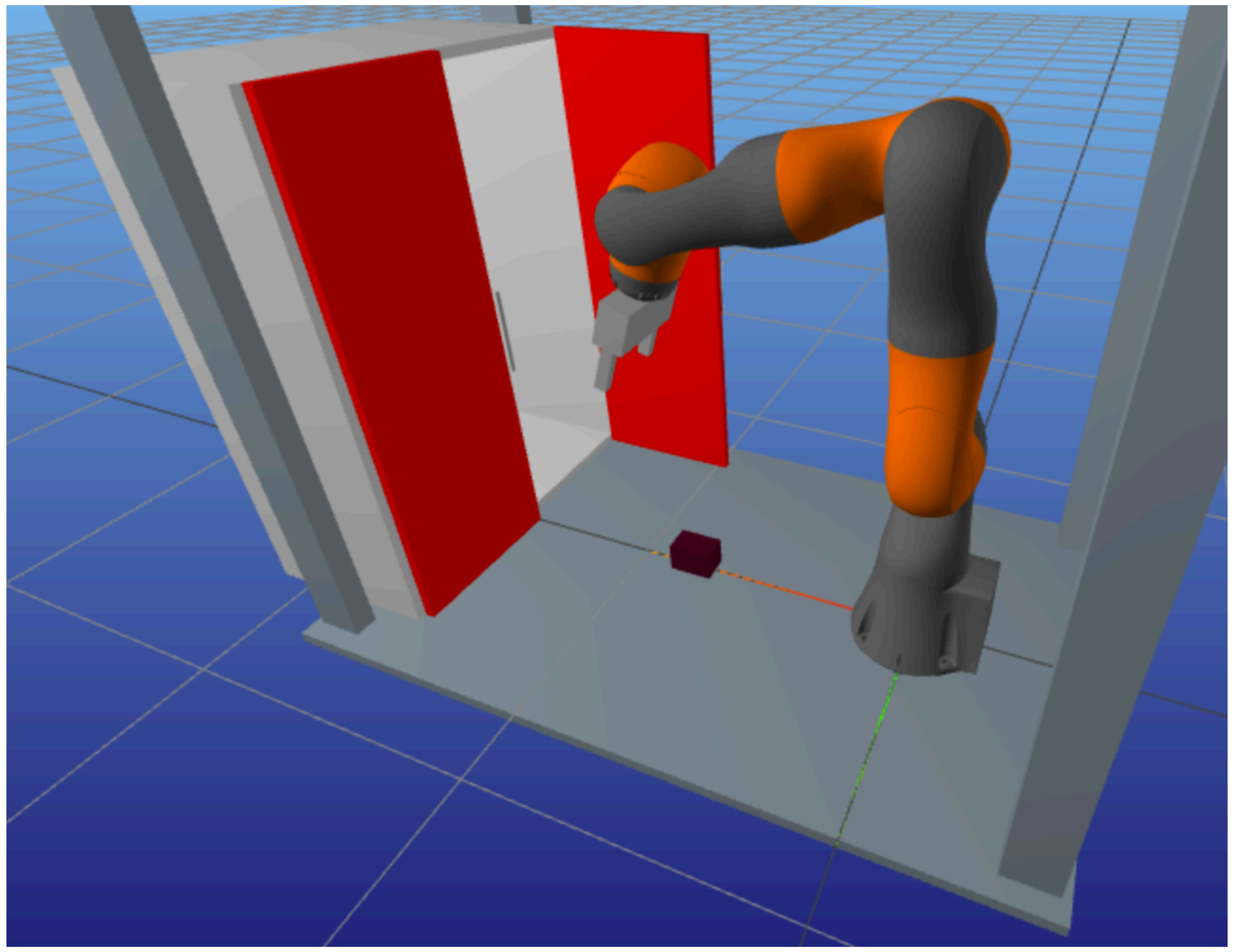




# TAMP Challenges



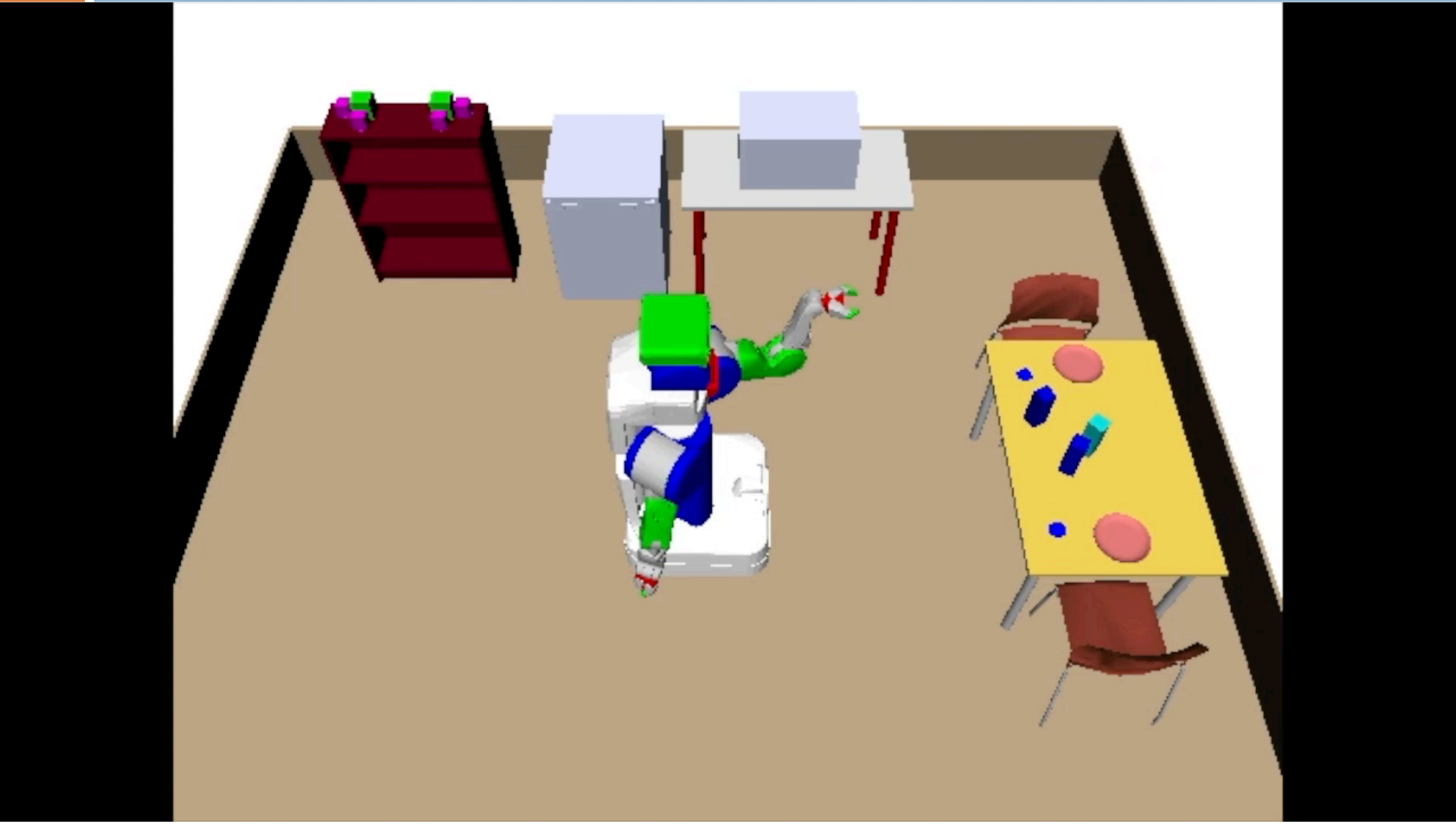
- **Automated robotic planning**
  - Robot only endowed with a **model** of the world
  - In contrast to **preprogramming** or **prediscretizing**
- **Inherits challenges** of both motion & AI planning
  - **Continuous** and **combinatorially** large spaces



- Continuous constraints **limit high-level strategies**
  - Kinematic reachability
  - Joint limits & collisions
- **Long horizons**

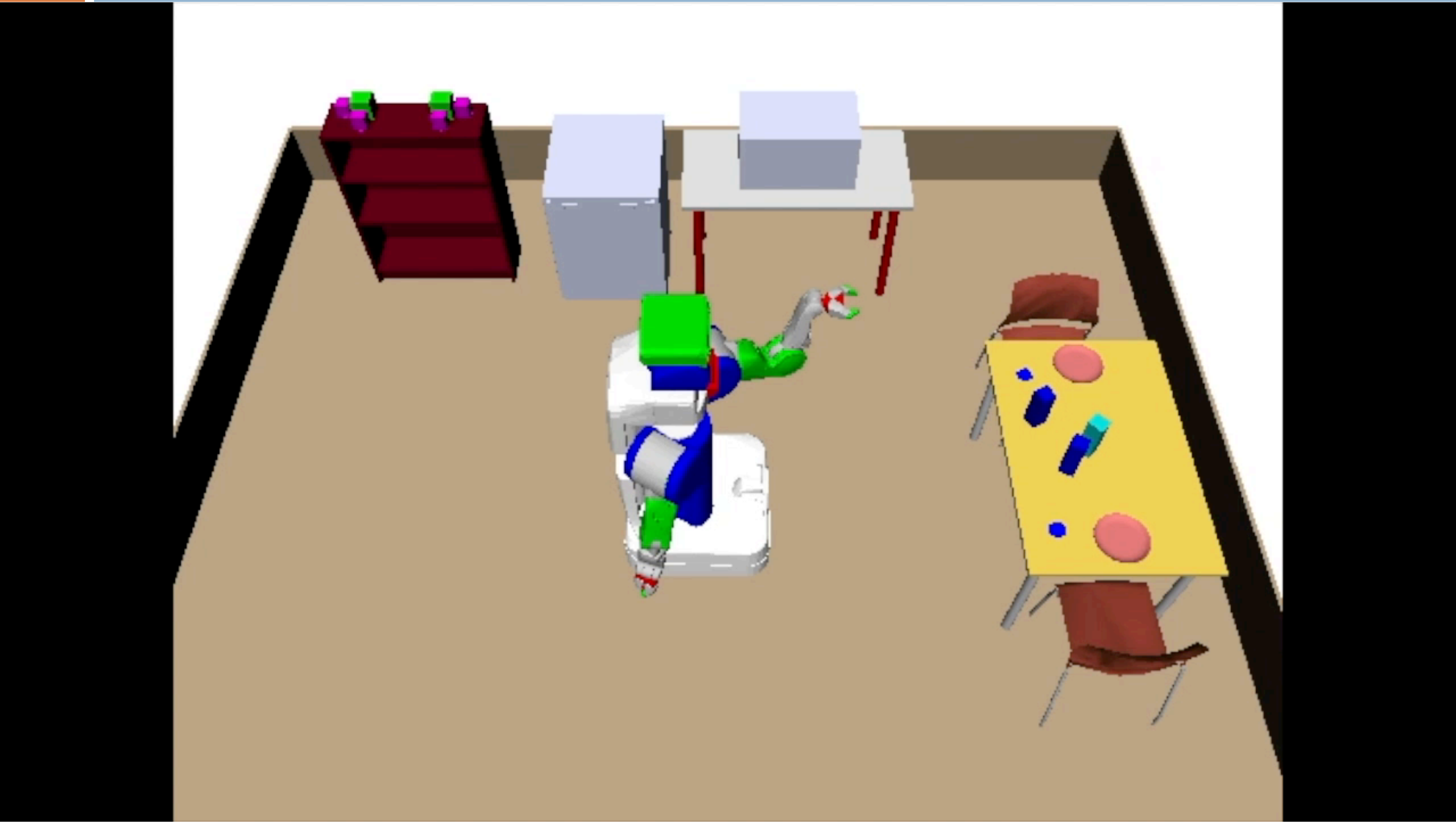
# PR2 Mobile Manipulation





# PR2 Mobile Manipulation





# Survey of Approaches

# Major Research Questions



- **What is a TAMP problem?**
- How do **describe** TAMP problems both in terms of:
  - Theoretical representation?
  - Operable forms consumable by an algorithm?

- How do we design algorithms that:
  - are **general-purpose**?
  - are empirically **efficient**?
  - have theoretical **guarantees**?
    - completeness & optimality
  - produce **low-cost** solutions?

# Taxonomy of Existing Approaches



- TAMP is **interdisciplinary** - intersection of AI and robotic planning
  - Most approaches stem from one community

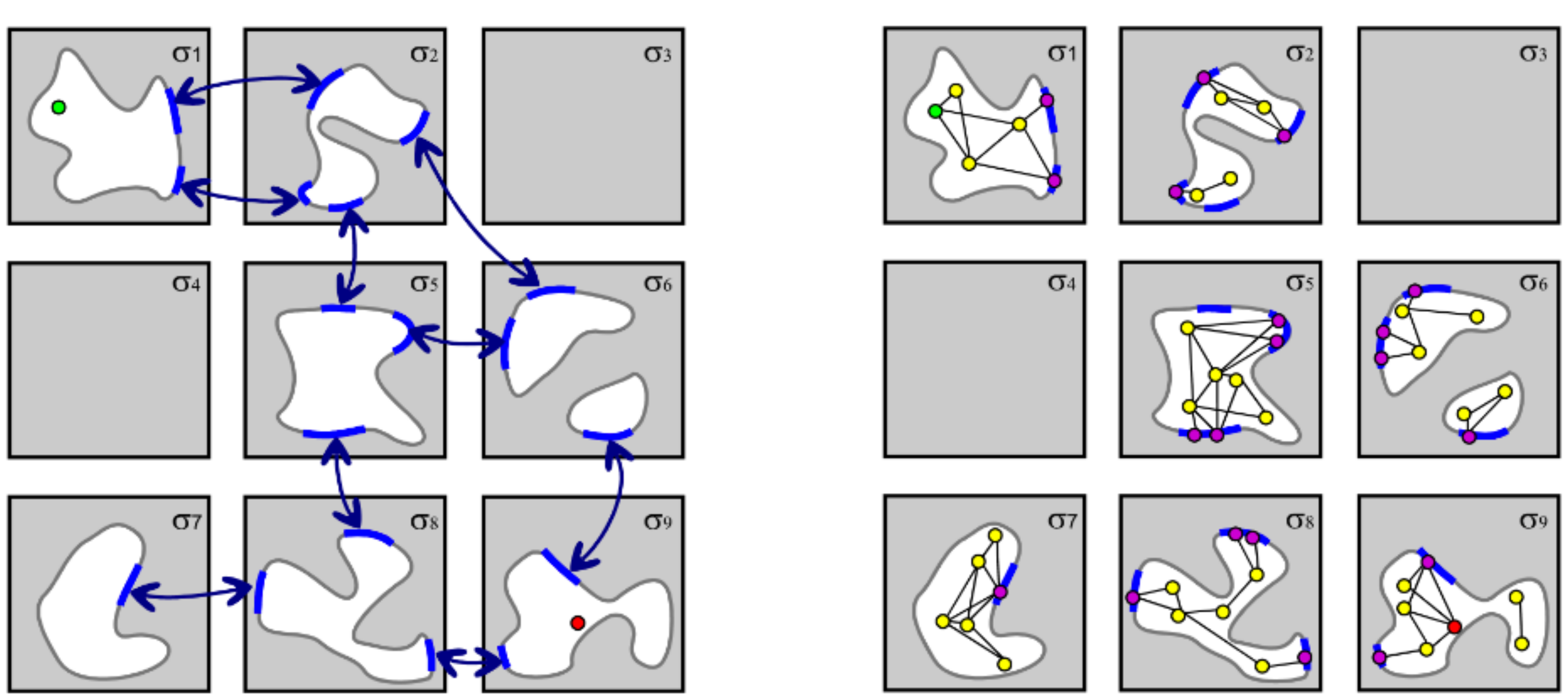
- **Extending motion planning**
  - Multi-modal motion planning
  - Optimization and constraint satisfaction

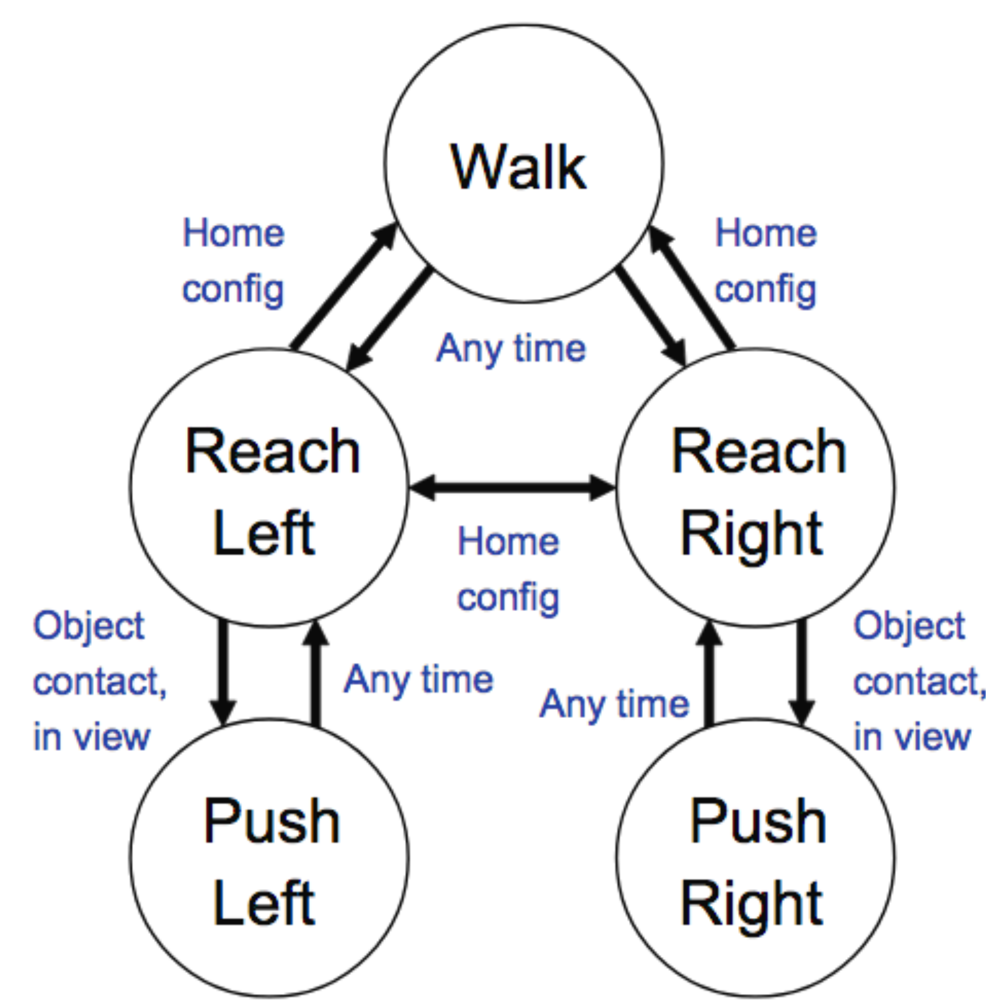
- **Extending AI planning**
  - Semantic attachments
  - Task and motion interface

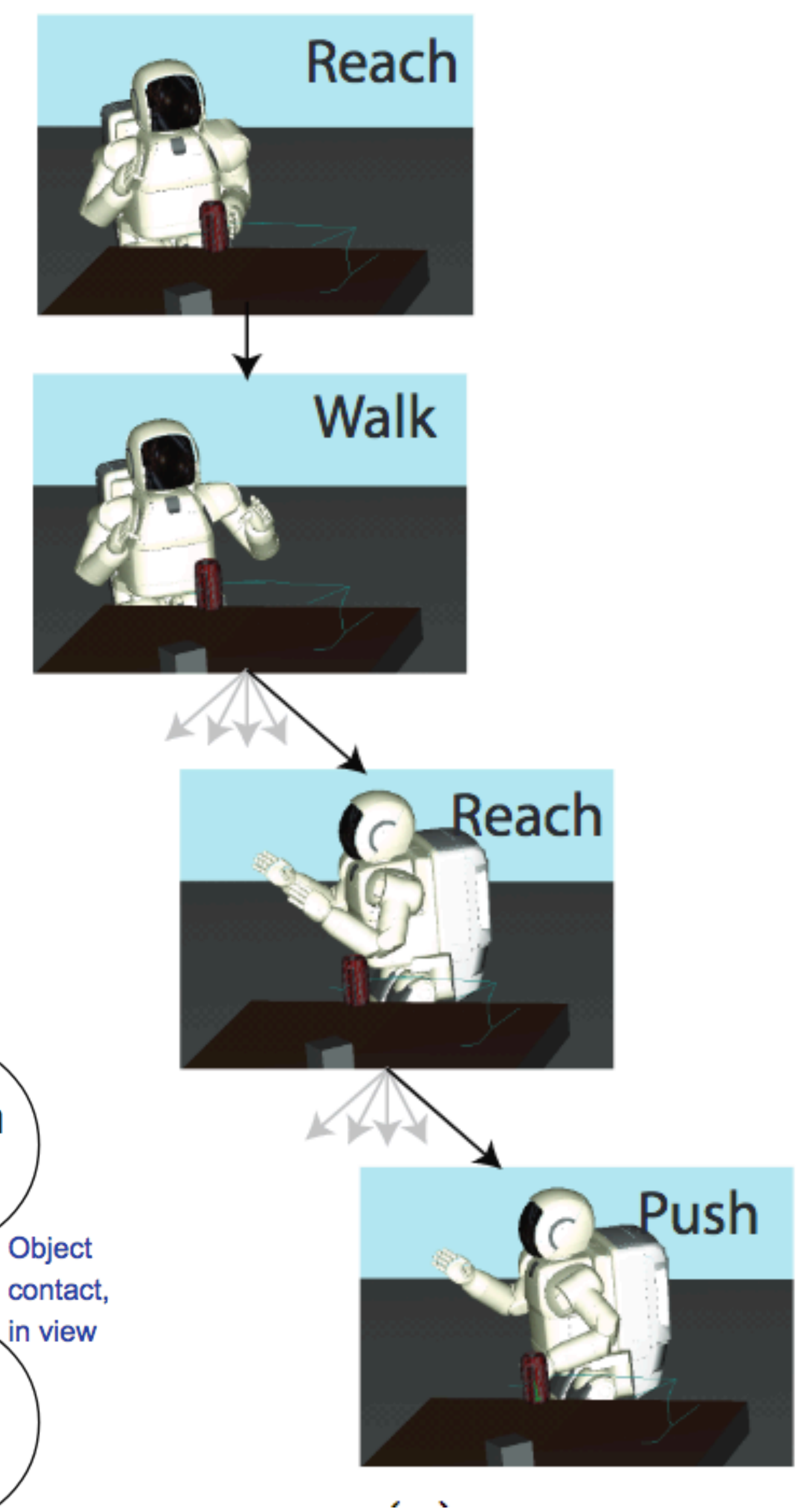
# Multi-Modal Motion Planning



- Sequence of motion planning problems through adjacent **modes** (motion constraints)
  - *[Alami & Siméon, Hauser & Latombe, Barry, Vega-Brown & Roy]*
- **Example modes** - hand empty, holding, pushing, …



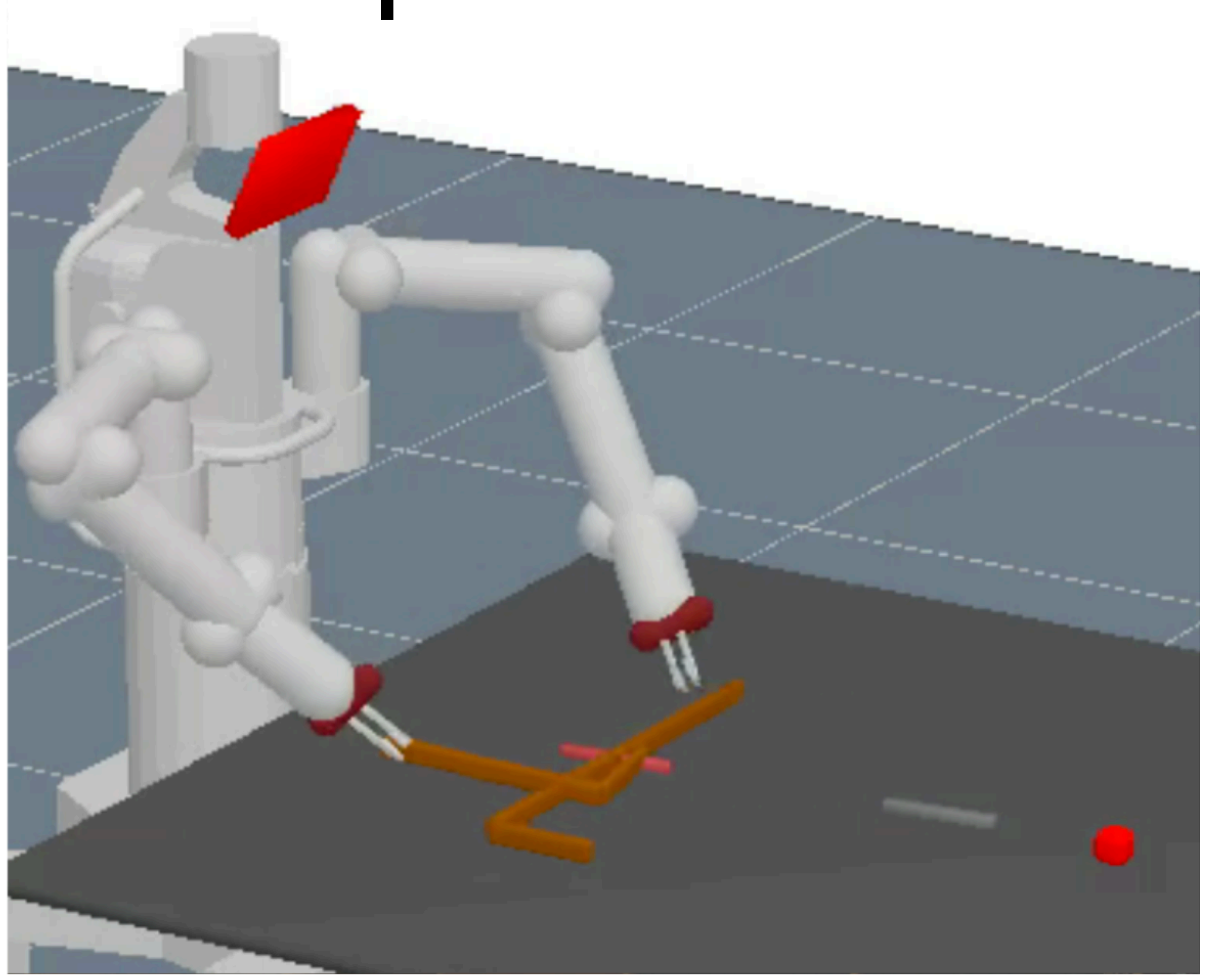



# Mathematical Programming



- Discrete search over sequences of **mode switches**
- Solve for values of continuous variables
  - Discretized **constraint satisfaction problem** (CSP) *[Lozano-Perez, Kaelbling]*
  - **Nonlinear constrained optimization** *[Toussaint]*
- Optimization-based motion planning (11/08/18)



$$
\begin{aligned}
\min_{x, a_{1:K}, s_{1:K}} \quad & \int_0^T f_{\text{path}}(\bar{x}(t))\, dt + f_{\text{goal}}(x(T)) \\
\text{s.t.} \quad & x(0) = x_0,\ h_{\text{goal}}(x(T)) = 0,\ g_{\text{goal}}(x(T)) \le 0, \\
& \forall t \in [0, T]:\ h_{\text{path}}(\bar{x}(t), s_{k(t)}) = 0, \\
& \qquad\qquad\quad g_{\text{path}}(\bar{x}(t), s_{k(t)}) \le 0 \\
& \forall k \in \{1, .., K\}:\ h_{\text{switch}}(\hat{x}(t_k), a_k) = 0, \\
& \qquad\qquad\qquad g_{\text{switch}}(\hat{x}(t_k), a_k) \le 0, \\
& \qquad\qquad\qquad s_k \in \text{succ}(s_{k\text{-}1}, a_k)\ . \qquad (2)
\end{aligned}
$$

# Semantic Attachments



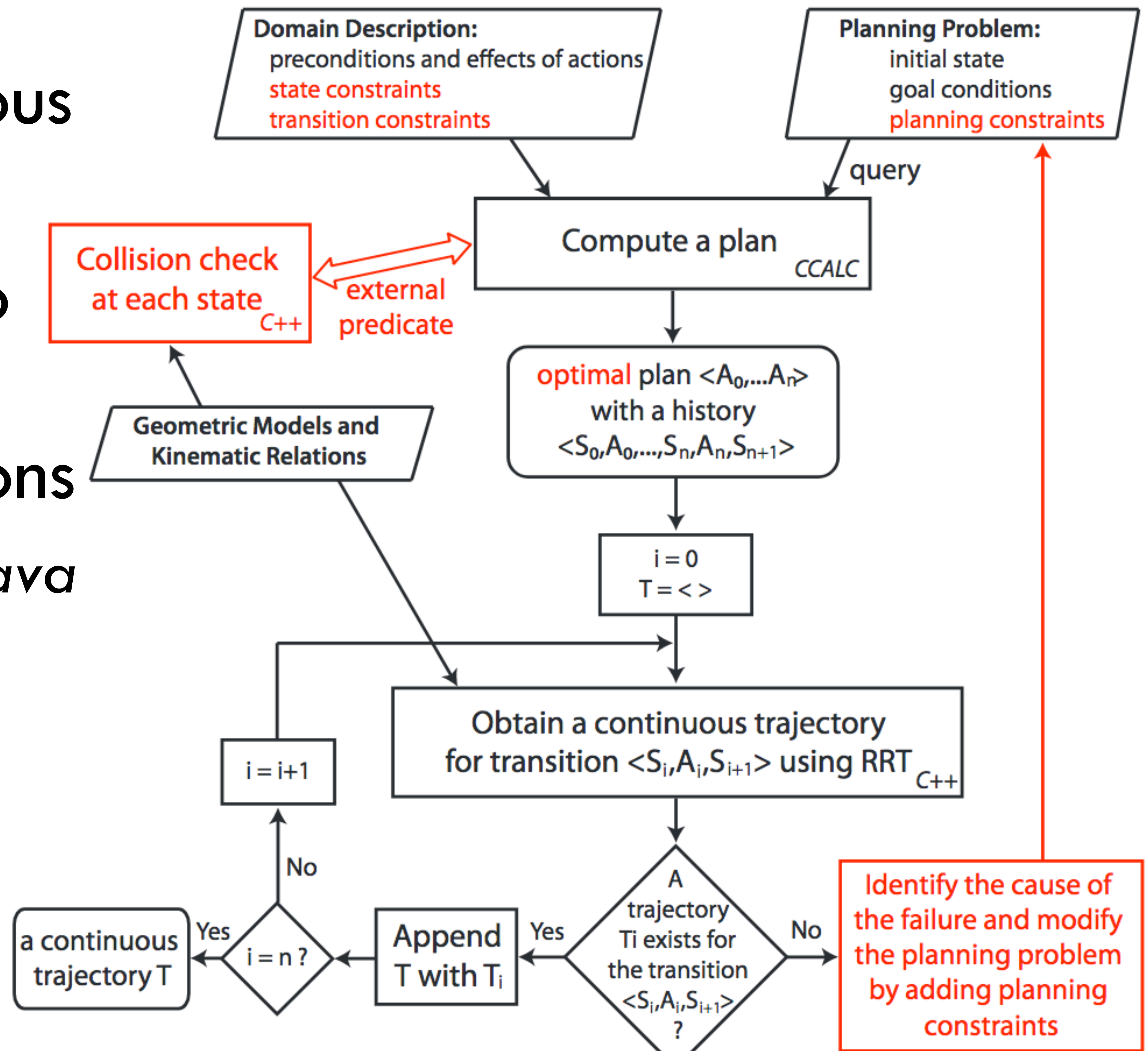
- Some preconditions & effects are defined using a **programming language** (e.g. C++, *.so library)
  - *[Dornhege et al., Gregory et al.]*
- **Example** - can the robot reach a placement?
- Evaluate attachments value during **forward search**

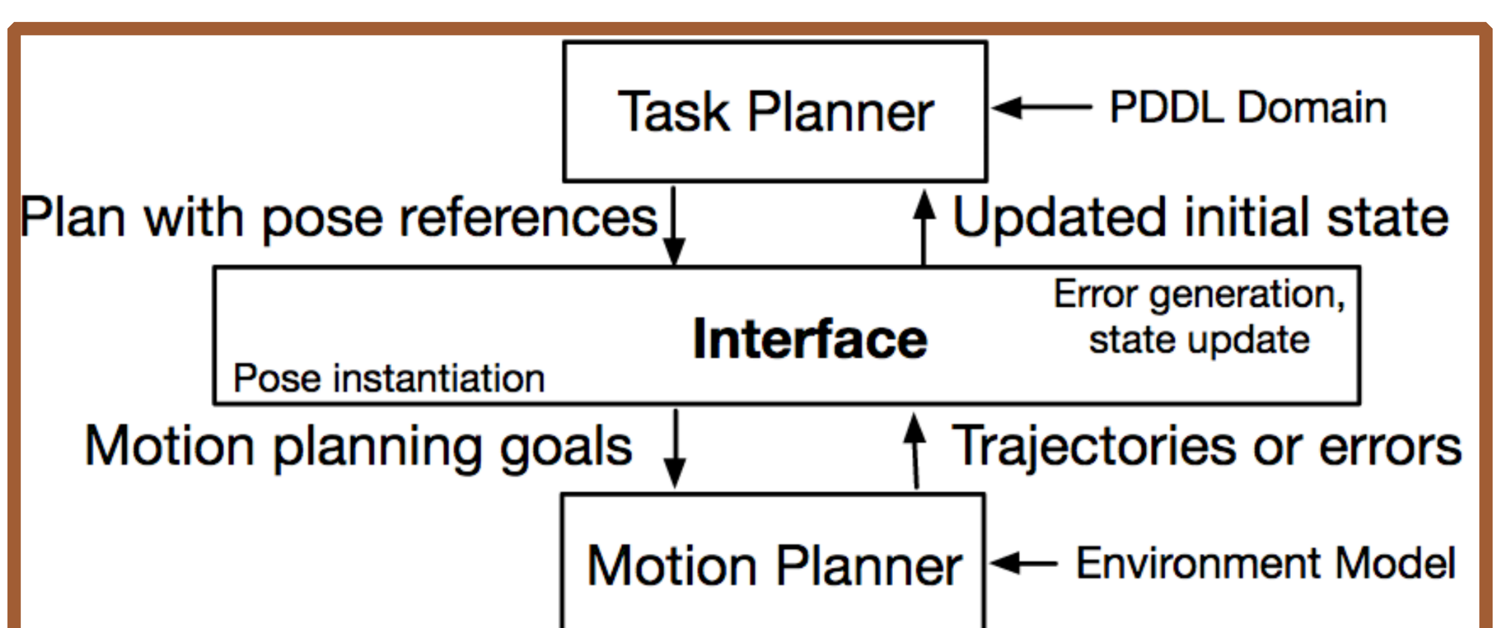
```
(:modules
  (putDown ?o – movable ?p – base ?g – grasp
    (q0) (q1) (q2) (q3) (q4) (q5) (q6)
    (x ?o) (y ?o) (z ?o)
    (yaw ?o) (pitch ?o) (roll ?o)
    effect [putDown@libTrajectory.so] )
(:action put-down
 :params (?o – movable ?p – base ?g – grasp)
 :condition (... ([checkPutDown ?o ?p ?g] ))
 :effect (and (on ?o ?p) (handempty)
  (not (holding ?o ?g)) ([putDown ?o ?p ?g] )
```

# Task & Motion Interface



- Maintain **separate** discrete & continuous descriptions
- Design interface to **communicate** between descriptions
  - *[Erdem et al., Srivastava et al., Dantam et al.]*



# Drawbacks



- **Multi-modal motion planning** - brute-force search through a high-dimensional space
- **Optimization and constraint satisfaction** - substantial backtracking when infeasible subproblems

- **Semantic attachments** - restricted to problems with finitely many actions
- **Task & motion interface** - lack of modularity

- Overall, **no general-purpose, flexible framework** for planning in a variety of TAMP problems

# PDDLStream

*[Garrett, Lozano-Perez, Kaelbling]*

# State and Action Representation



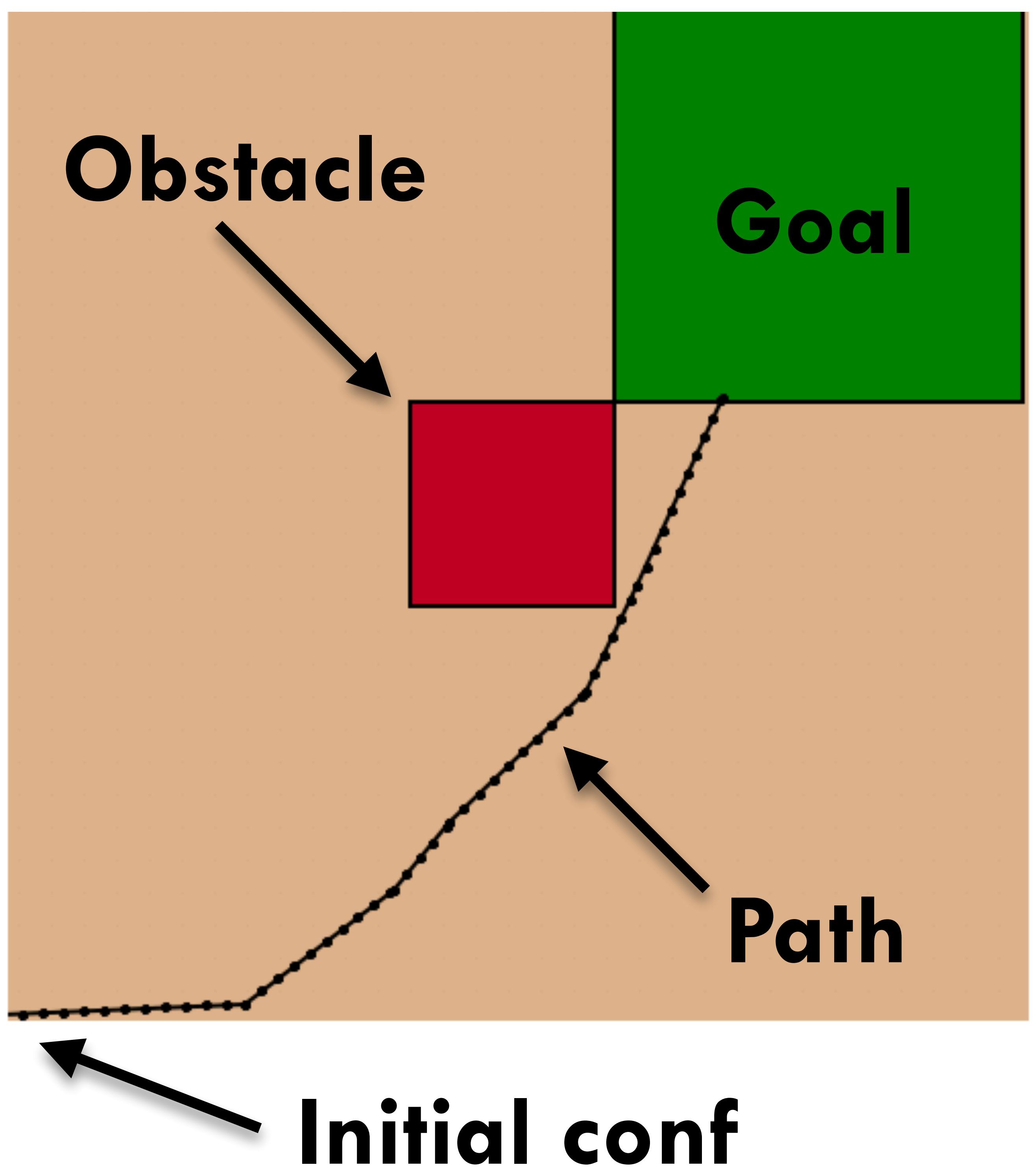
- **Requirements**
  - **Lifted** - actions are parameterized
    - **Objects** instantiate predicates & actions
    - Can encode an **infinite set** of action instances
  - **Factored** - state is a collection of variables that can change independently

- Extend Planning Domain Description Language (**PDDL**)
  - Primarily just for **standardization** purposes
  - Other possible representations:
    - STRIPS, factored transition system, Prolog, …

# PDDLStream Motion Planning



- Model **motion planning** for a point robot using **PDDL**
- **Goal:** point robot within the **green** goal region




```
(define (domain motion-planning)
 (:predicates
    (Conf ?q) (Region ?r)
    (Connected ?q1 ?q2) (Contained ?q ?r)
    (AtConf ?q) (In ?r))

  (:action move
    :parameters (?q1 ?q2)
    :precondition (and (Connected ?q1 ?q2)
                       (AtConf ?q1))
    :effect (and (AtConf ?q2)
                 (not (AtConf ?q1))))

  (:derived (In ?r)
    (exists (?q) (and (Contained ?q ?r)
                      (AtConf ?q))))
```

**Continuous!**

# Motion Planning Initial & Goal



- **Static predicates** - value is constant over time

```
(Conf ?q)  (Region ?r) (Connected ?q1 ?q2)  (Contained ?q ?r)
```

- **Fluent predicates** - value may change over time due to **action effects.** Intuitively, comprise **state variables**

```
(AtConf ?q)
```

- **Derived predicates** - value is a logical formula

```
(:derived (In ?r)
    (exists (?q) (and (Contained ?q ?r) (AtConf ?q)))))
```

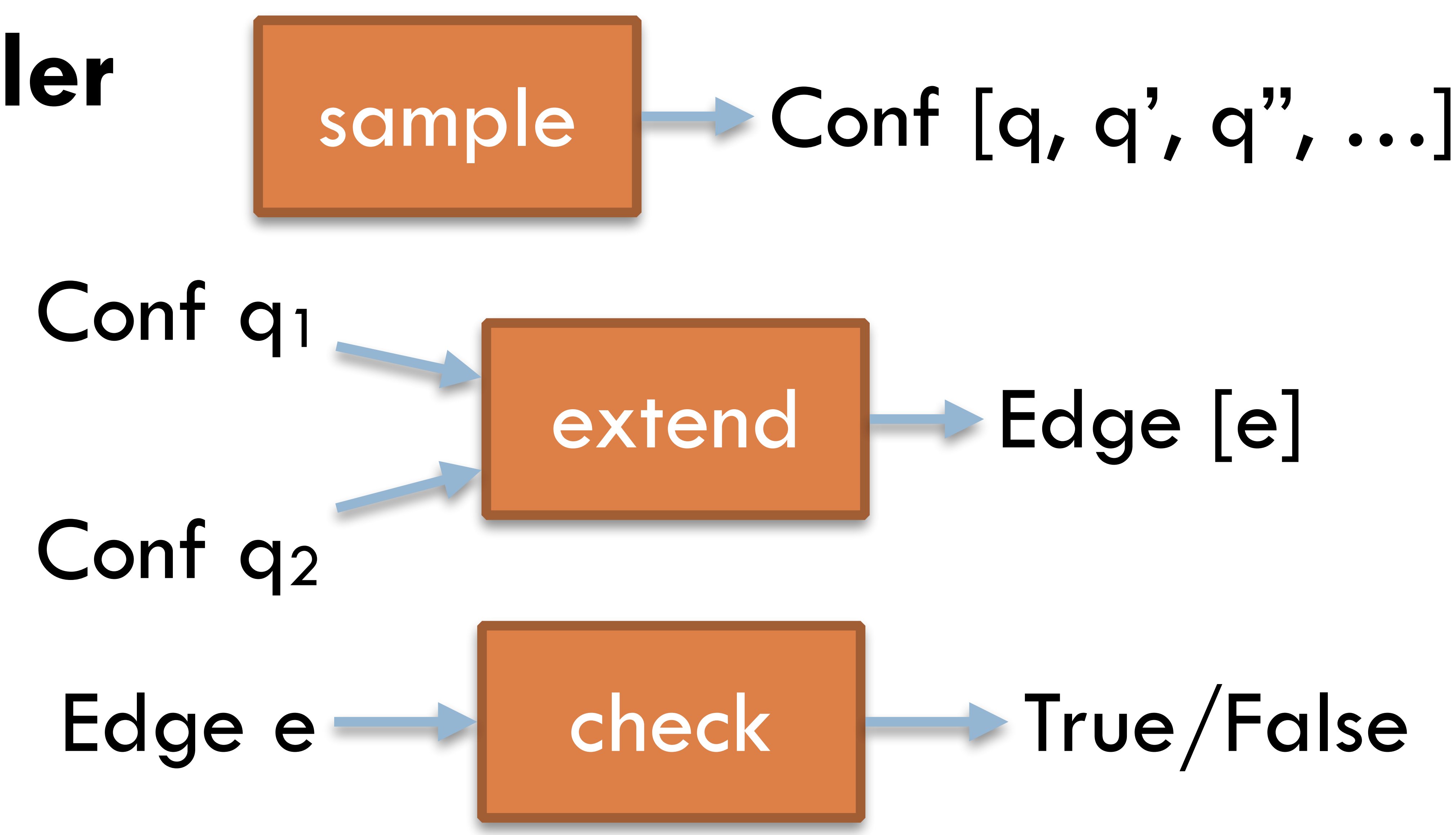
- **Initial state:** [(Conf, [0 0]), (AtConf, [0 0]), (Region, green), (Region, env)]
- **Goal formula:** (In green)

# Representing Infinitely Many Objects



- How do encode & use an **infinite set** of configurations?
  - How do we do this in a **domain-independent** fashion?

- Motion planning does this by **abstracting** out the following procedures as **meta-parameters**
  - **Configuration sampler**
  - **Extension function**
  - **Collision checker**



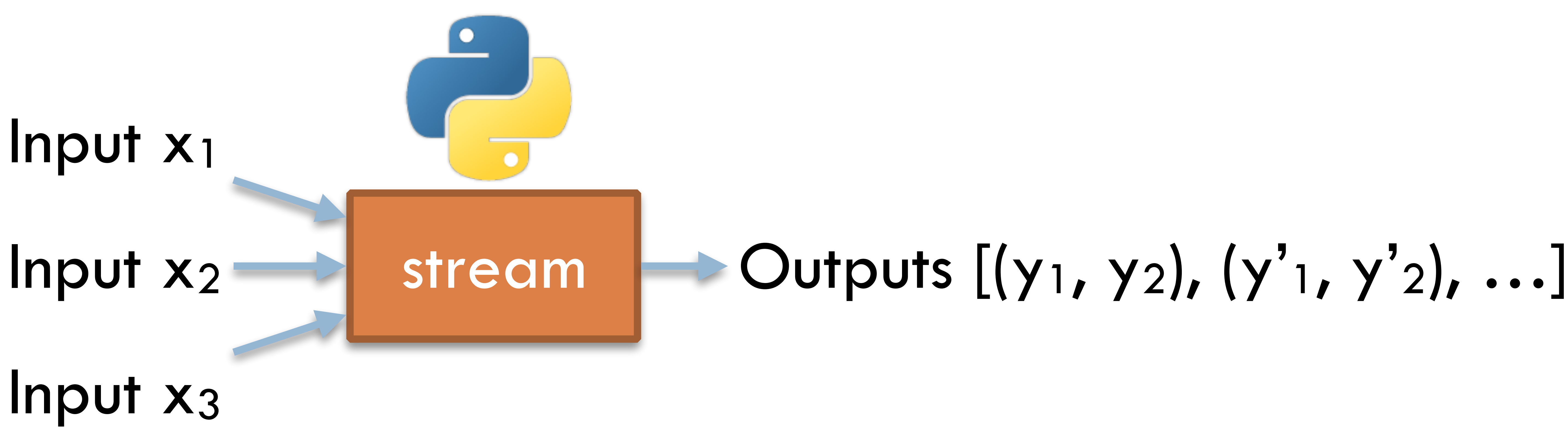
# Stream: a function to a generator



- **Requirements**
  - Programmatic implementation
  - Supports infinite sequences
  - Compositional

```python
def stream(x1, x2, x3):
    i = 0
    while True:
        y1 = i*(x1 + x2)
        y2 = i*(x2 + x3)
        yield (y1, y2)
        i += 1
```

- **Stream** - function from an **input object tuple** $(x_1, x_2, x_3)$ to a (potentially infinite) sequence of **output object tuples** $[(y_1, y_2), (y'_1, y'_2), \ldots]$



# Stream Certified Facts



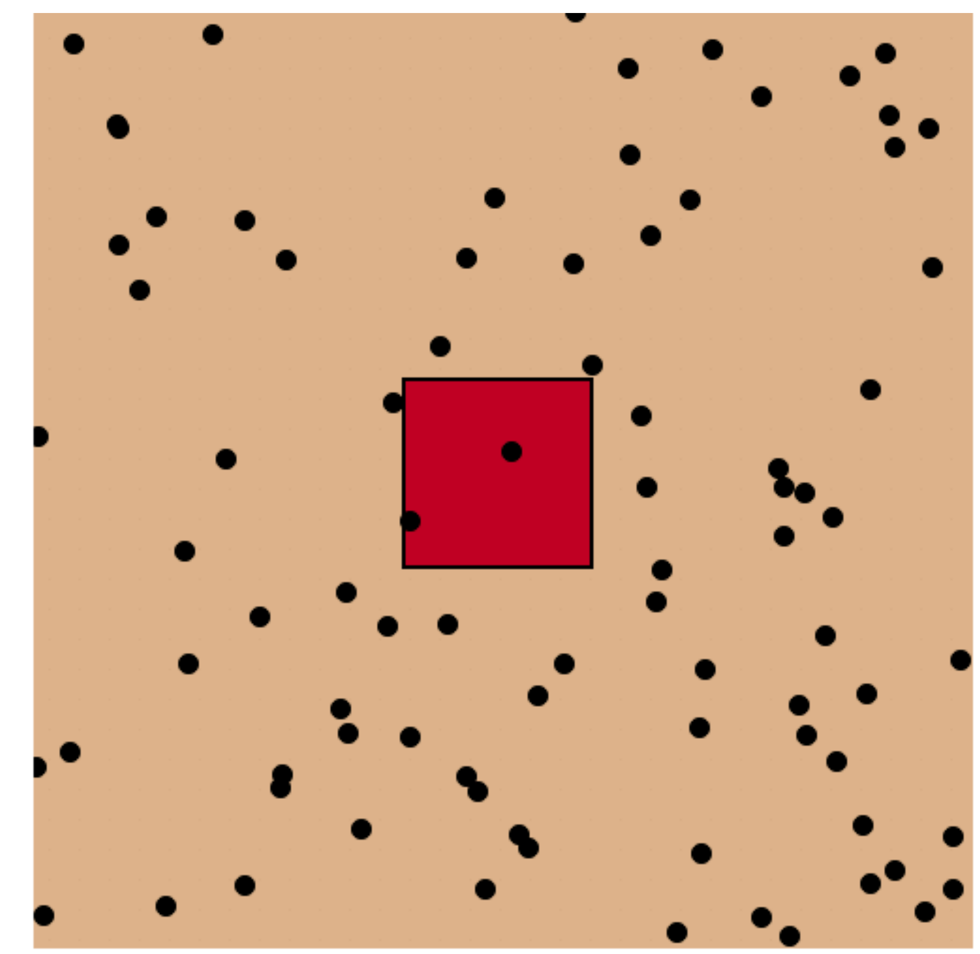
- Objects alone aren't helpful: **what do they represent?**
  - Communicate semantics using **predicates**!

- Augment stream specification with:
  - **Certified facts** - static facts that all **outputs** satisfy with their corresponding **inputs**
    - e.g. poses sampled from a region are within it
  - **Domain facts** - static facts declaring legal inputs
    - e.g. only configurations can be connected

# Motion Planning Streams

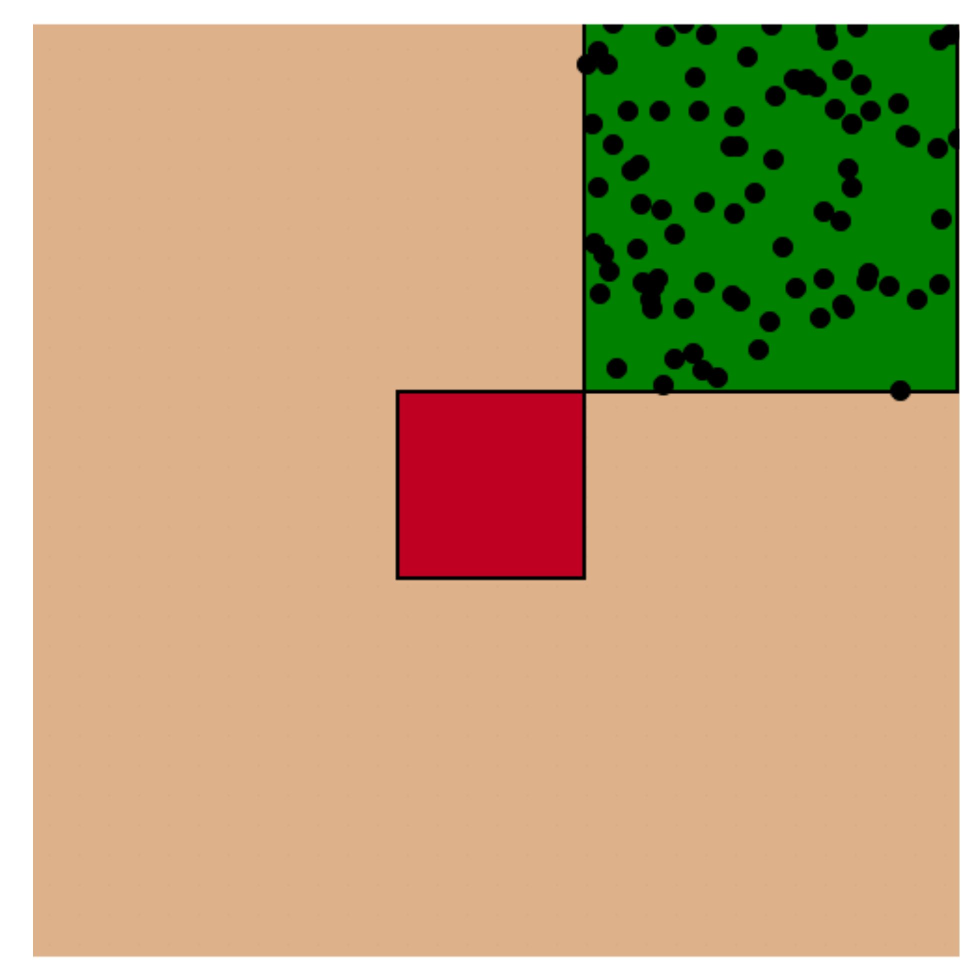


- Samples an axis-aligned region **uniformly at random**
- Generates arbitrarily many confs



```
(:stream sample-region
   :inputs (?r)
   :domain (Region ?r)
   :outputs (?q)
   :certified (and (Conf ?q)
                   (Contained ?q ?r)))
```

```python
def sample_region(r):
    (lower, upper) = REGIONS[r]
    while True:
        q = np.random.uniform(lower, upper)
        yield (q,)
```



Region r → sample-region → Conf [(q), (q'), …]

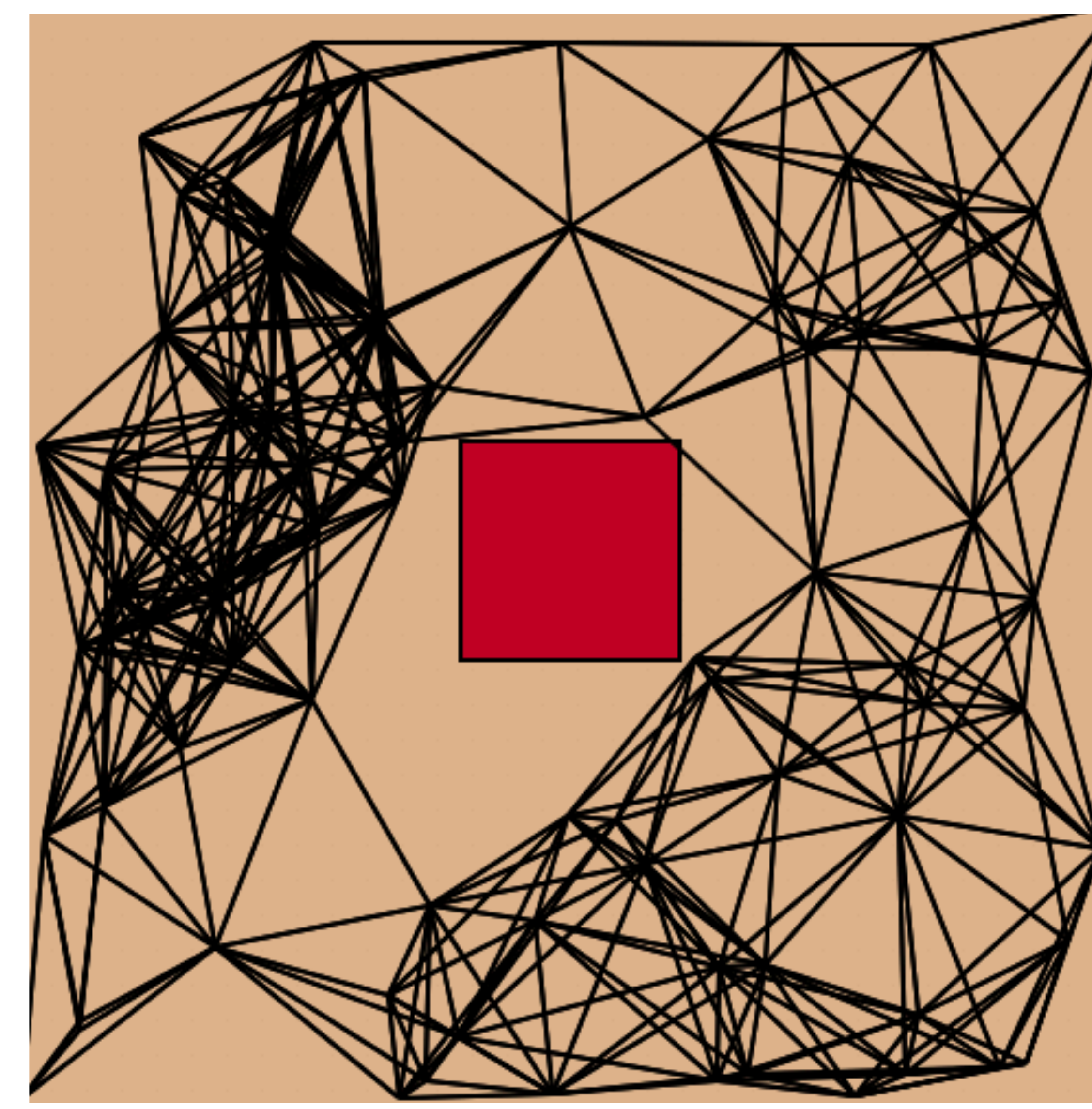
# Motion Planning Streams



- **Test stream:** a stream without outputs
- If returns the **empty tuple**, its certified facts are true

```
(:stream connect
  :inputs (?q1 ?q2)
  :domain (and (Conf ?q1) (Conf ?q2))
  :outputs ()
  :certified (Connected ?q1 ?q2)))
```

```python
def connect(q1, q2):
    if (np.linalg.norm(q2 - q1) < MAX_DIST) \
          and not segment_collision(q1, q2):
        yield tuple()
```



# PDDLStream = PDDL + Streams



- **Extension of PDDL** to support the specification of **blackbox procedures as streams**
  - *[Garrett, Lozano-Perez, Kaelbling]*
- The **true initial state** is the set of **all static facts** that can be certified by streams (may be **infinitely large)**
- Static facts restrict **domain of action parameters**

```
(:action move
   :parameters (?q1 ?q2)
   :precondition (and (Connected ?q1 ?q2)
                      (AtConf ?q1))
   :effect (and (AtConf ?q2)
                (not (AtConf ?q1))))
(:derived (In ?r)
   (exists (?q) (and (Contained ?q ?r)
                     (AtConf ?q))))
```

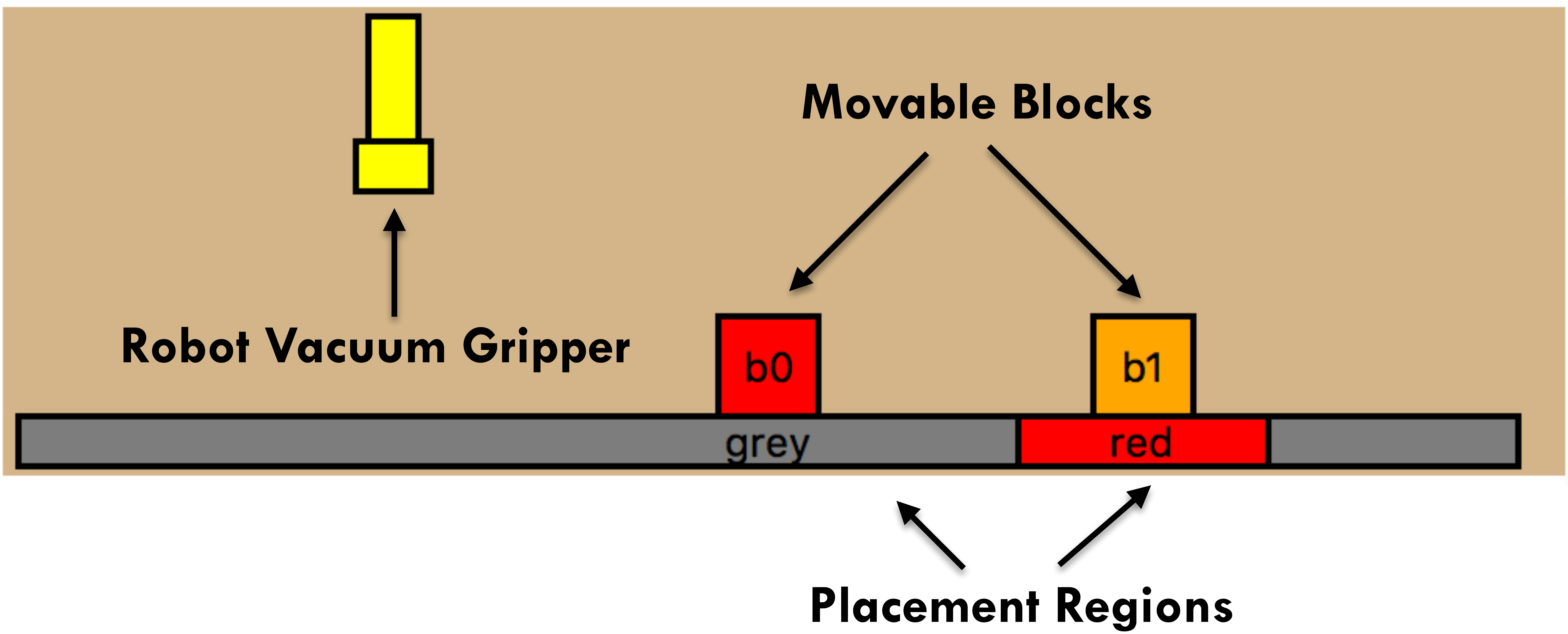
Certified by **connect**

Certified by **sample-region**

# 2D Pick-and-Place Example



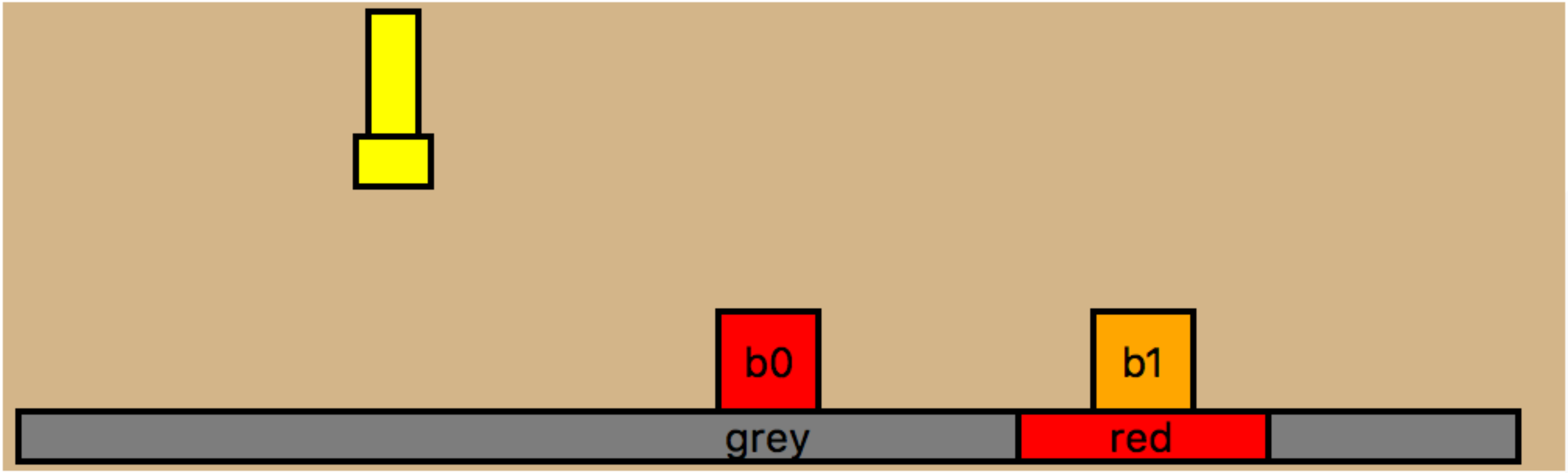
- **Goal**: block **b0** within the **red** region
- Robot and block poses are continuous (x, y) pairs
- Block **b1** obstructs the placement of **b0**



# 2D Pick-and-Place Solution



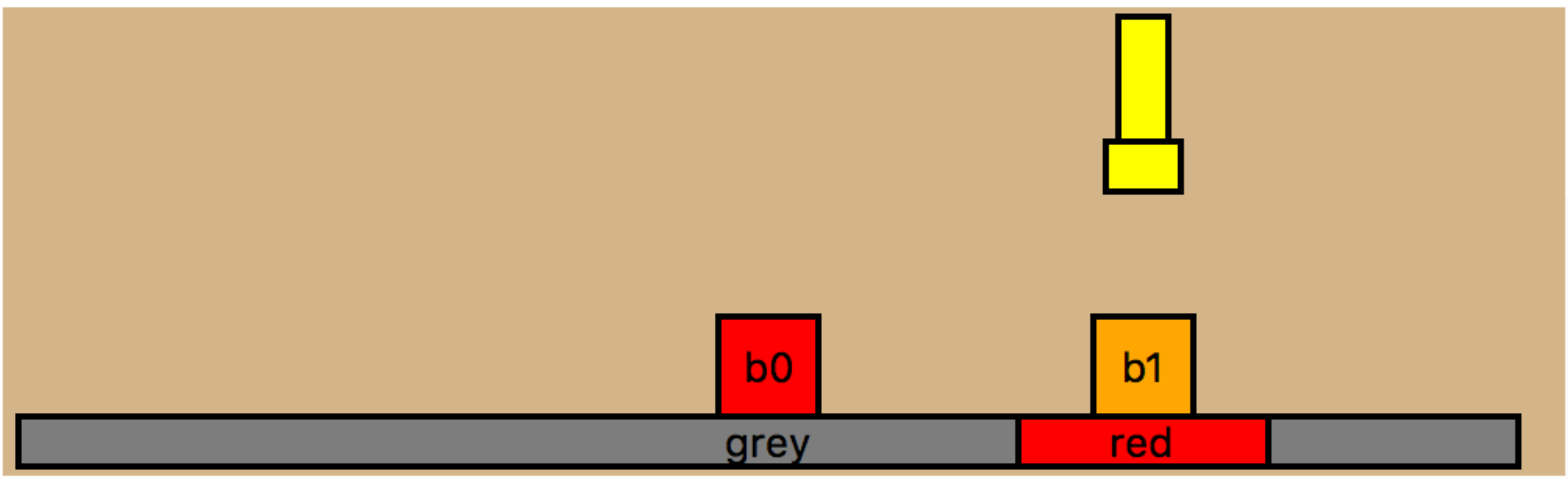
- One (of many) possible solutions
  - move, pick b1, move, place b1, move, pick b0, move, place b0



# 2D Pick-and-Place Solution



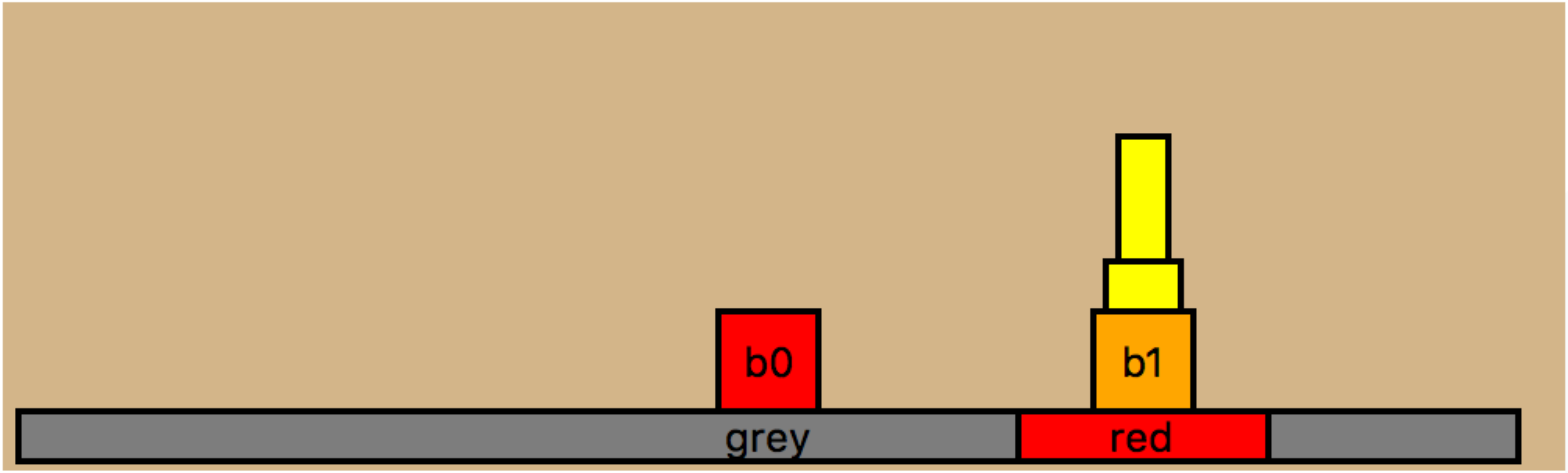
- One (of many) possible solutions
  - move, pick b1, move, place b1, move, pick b0, move, place b0



# 2D Pick-and-Place Solution



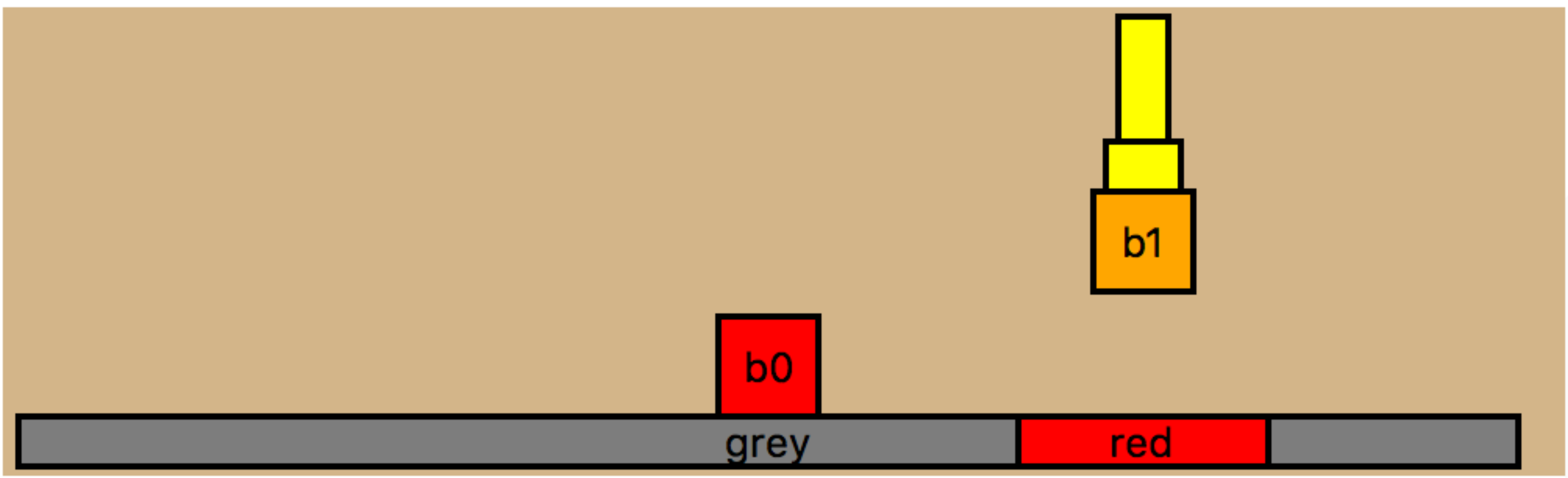
- One (of many) possible solutions
  - move, pick b1, move, place b1, move, pick b0, move, place b0



# 2D Pick-and-Place Solution



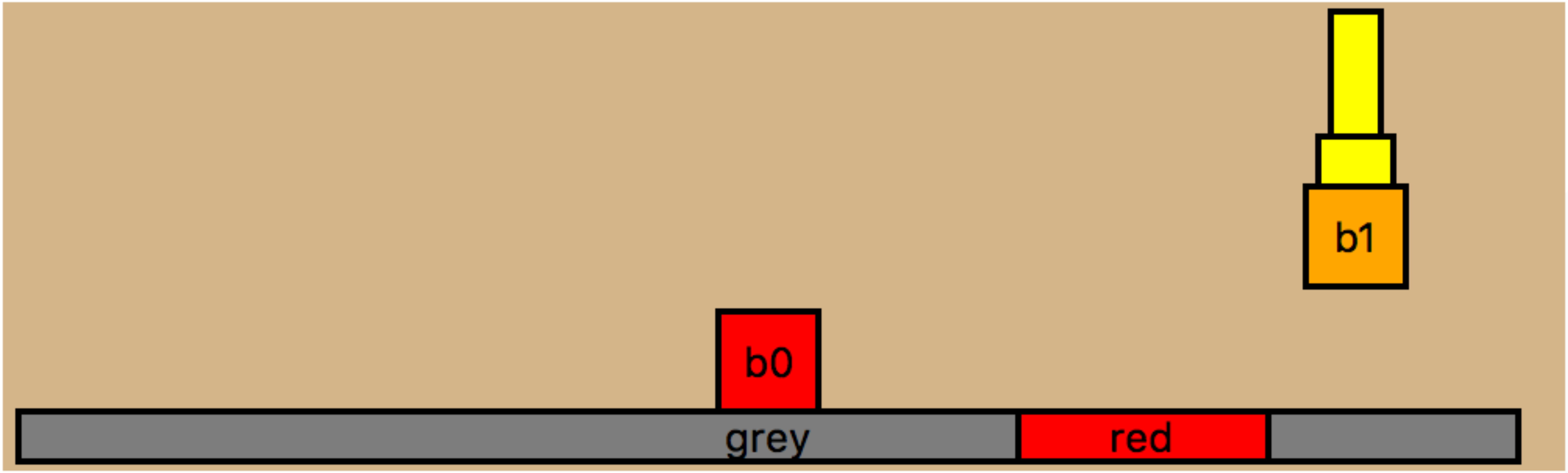
- One (of many) possible solutions
  - move, pick b1, move, place b1, move, pick b0, move, place b0



# 2D Pick-and-Place Solution



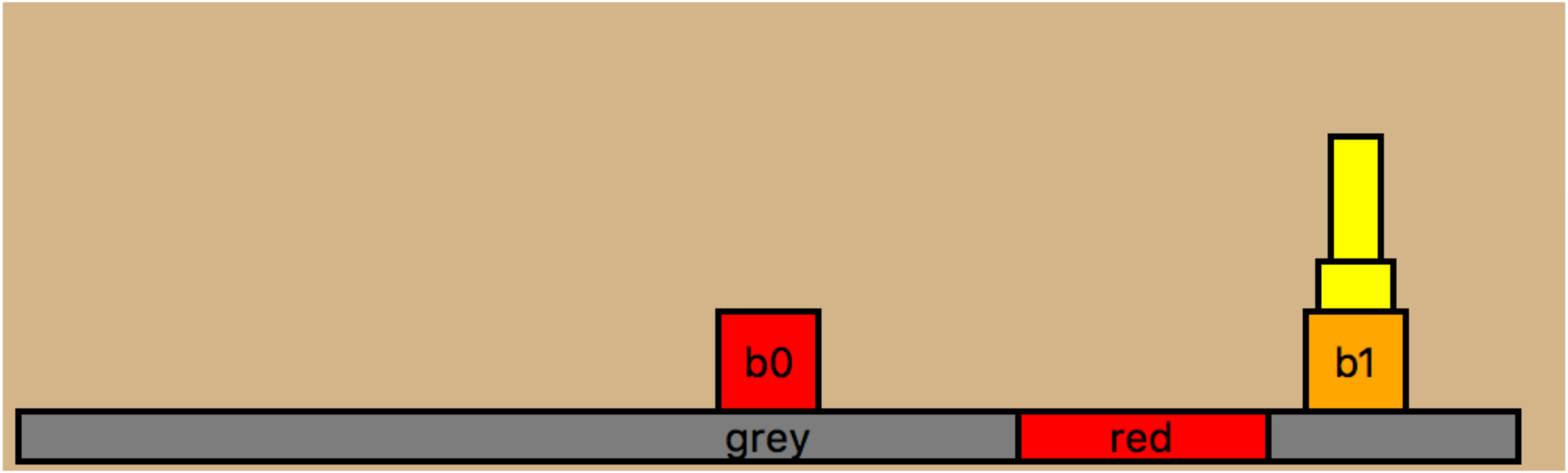
- One (of many) possible solutions
  - move, pick b1, move, place b1, move, pick b0, move, place b0



# 2D Pick-and-Place Solution



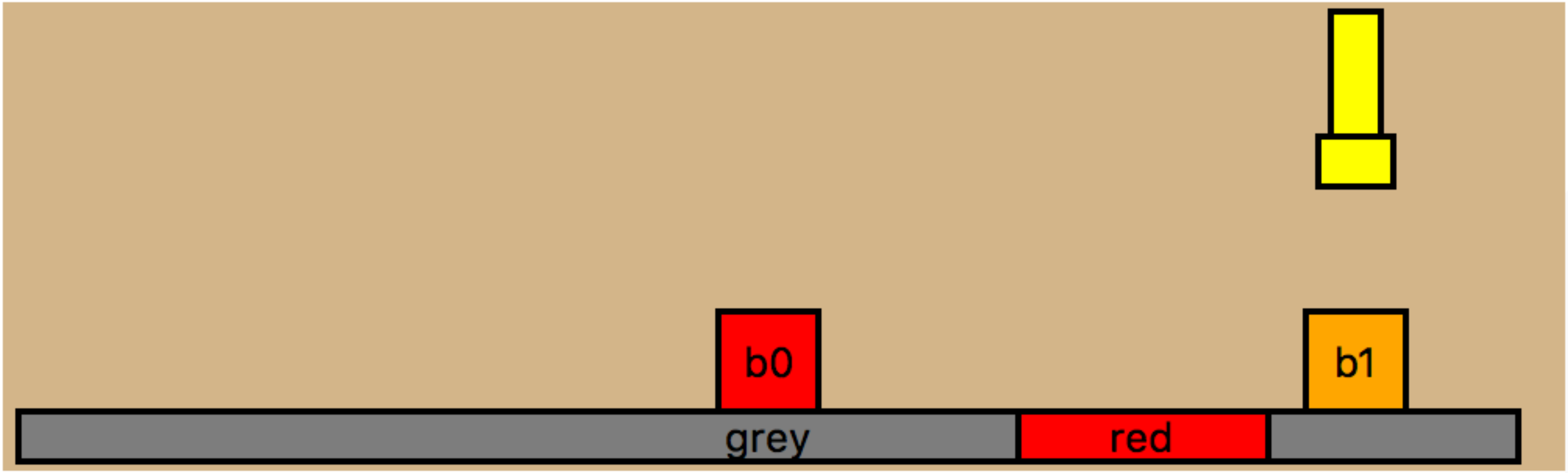
- One (of many) possible solutions
  - move, pick b1, move, place b1, move, pick b0, move, place b0



# 2D Pick-and-Place Solution



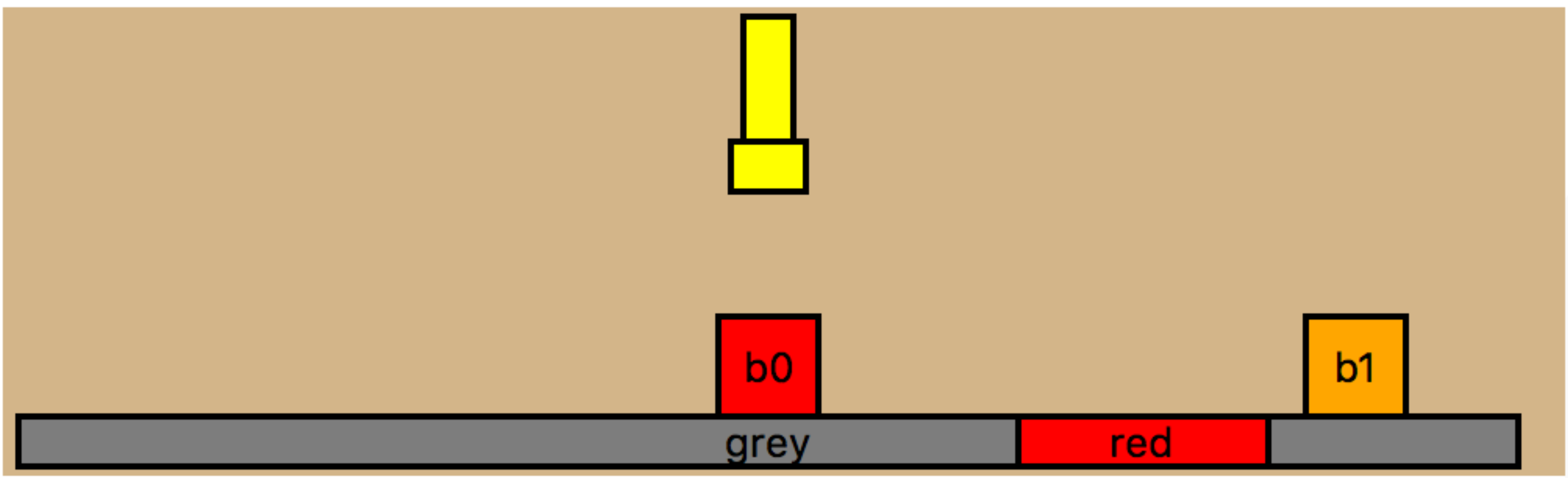
- One (of many) possible solutions
  - move, pick b1, move, place b1, move, pick b0, move, place b0



# 2D Pick-and-Place Solution



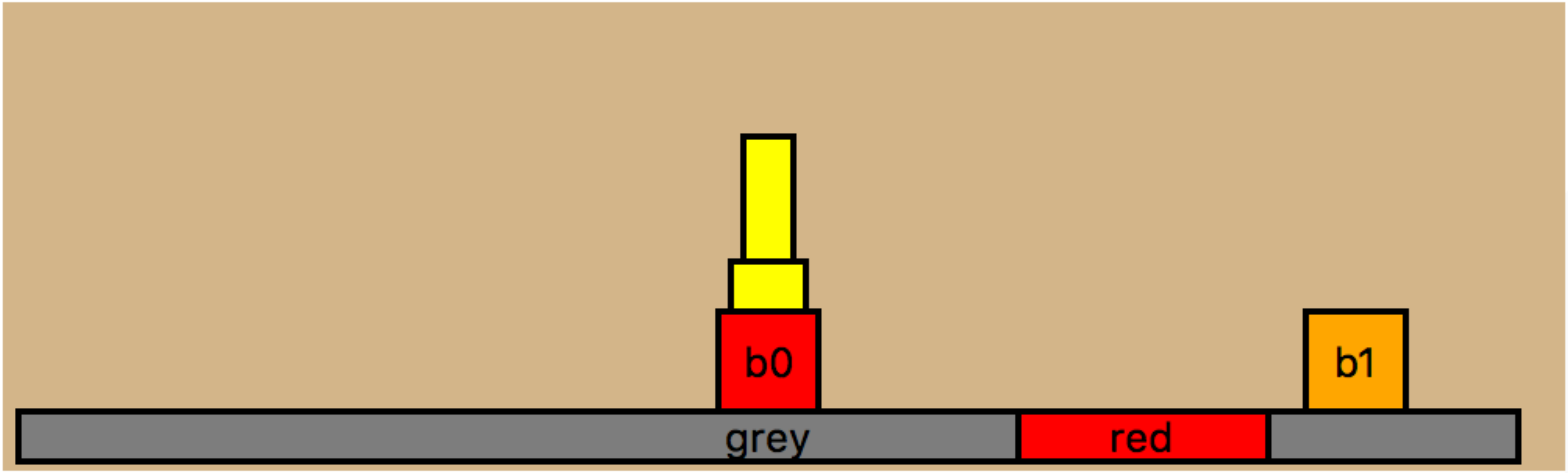
- One (of many) possible solutions
  - move, pick b1, move, place b1, move, pick b0, move, place b0



# 2D Pick-and-Place Solution



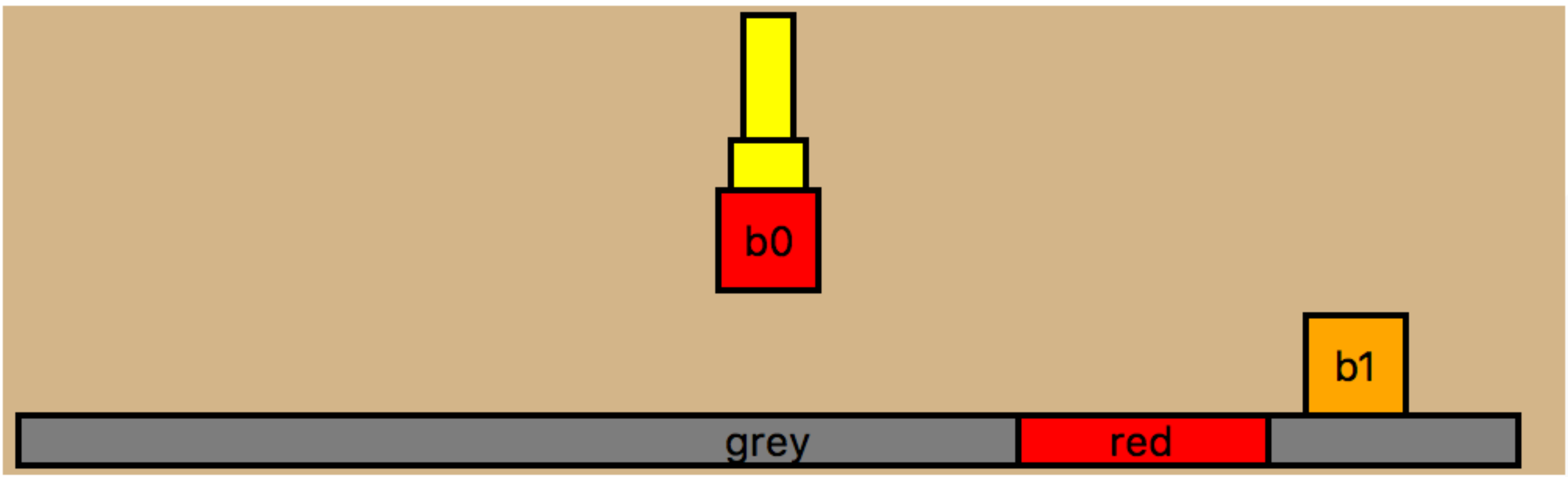
- One (of many) possible solutions
  - move, pick b1, move, place b1, move, pick b0, move, place b0



# 2D Pick-and-Place Solution



- One (of many) possible solutions
  - move, pick b1, move, place b1, move, pick b0, move, place b0



# 2D Pick-and-Place Solution



- One (of many) possible solutions
  - move, pick b1, move, place b1, move, pick b0, move, place b0

# 2D Pick-and-Place Solution



- One (of many) possible solutions
  - move, pick b1, move, place b1, move, pick b0, move, place b0

# 2D Pick-and-Place Actions



- Typical STRIPS action description except that arguments are mostly **continuous**!

```
(:action move
  :parameters (?q1 ?t ?q2)
  :precondition (and (Motion ?q1 ?t ?q2) (AtConf ?q1))
  :effect (and (AtConf ?q2) (not (AtConf ?q1))))

(:action pick
  :parameters (?b ?p ?q)
  :precondition (and (Kin ?b ?q ?p)
                     (AtConf ?q) (AtPose ?b ?p) (HandEmpty))
  :effect (and (Holding ?b)
               (not (AtPose ?b ?p)) (not (HandEmpty))))

(:action place
  :parameters (?b ?p ?q)
  :precondition (and (Kin ?b ?q ?p) (AtConf ?q) (Holding ?b))
  :effect (and (AtPose ?b ?p) (HandEmpty) (not (Holding ?b))))
```

# 2D Pick-and-Place Initial & Goal



- **Static predicates**

```
(Block ?b) (Region ?r) (Pose ?b ?p) (Conf ?q) (Traj ?t)
(Contained ?b ?p ?r) (Kin ?b ?q ?p) (Motion ?q1 ?t ?q2)
(CFree ?b1 ?p1 ?b2 ?p2)
```

- **Fluent predicates**

```
(AtPose ?b ?p) (AtConf ?q) (Holding ?b) (HandEmpty)
```

- **Derived predicates**

```
(:derived (In ?b ?r)
   (exists (?p) (and (Contained ?b ?p ?r) (AtPose ?b ?p))))
```

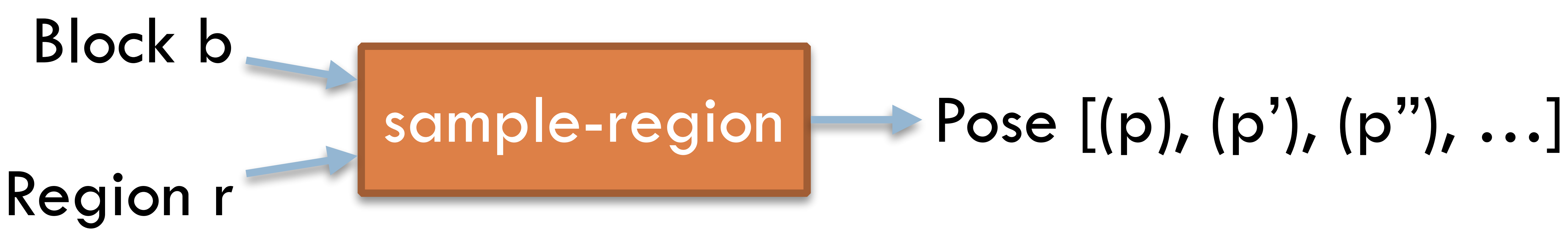
- **Initial state:** [(Conf, **[-7.5  5. ]**), (AtConf, **[-7.5  5. ]**), (HandEmpty), (Block, b0), (Block, b1), (Pose, b0, **[0. 0.]**), (Pose, b1, **[7.5 0. ]**), (AtPose, b0, **[0. 0.]**), (AtPose, b1, **[7.5 0. ]**), (Region, red), (Region, grey)]
- **Goal formula:** (In, b0, red)

# 2D Pick-and-Place Streams



```
(:stream sample-region
  :inputs (?b ?r)
  :domain (and (Block ?b) (Region ?r))
  :outputs (?p)
  :certified (and (Pose ?b ?p) (Contained ?b ?p ?r)))
```

```python
def sample_region(b, r):
  x_min, x_max = REGIONS[r]
  w = BLOCKS[b].width
  while True:
      x = random.uniform(x_min + w/2, x_max - w/2)
      p = np.array([x, 0])
      yield (p,)
```



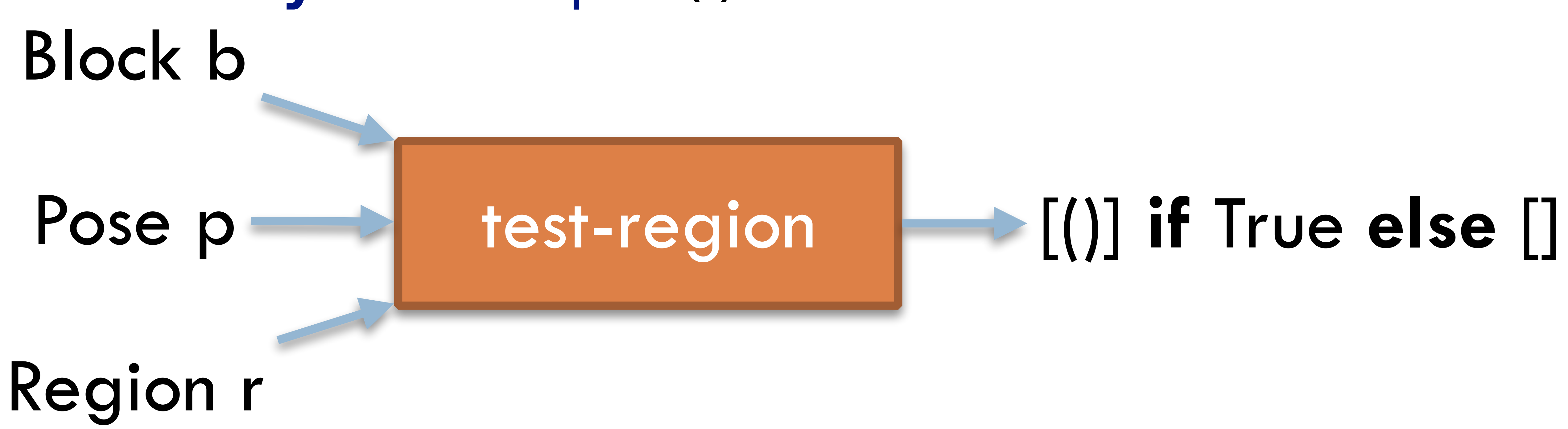
# 2D Pick-and-Place Streams



- **Multiple streams** can certify the same predicates

```
(:stream test-region
  :inputs (?b ?p ?r)
  :domain (and (Pose ?b ?p) (Region ?r))
  :outputs ()
  :certified (Contained ?b ?p ?r))
```

```python
def test_region(b, p, r):
  x, y = p
  w = BLOCKS[b].width
  x_min, x_max = REGIONS[r]
  if x_min <= (x - w/2) <= (x + w/2) <= x_max:
    yield tuple()
```



# 2D Pick-and-Place Streams



- **Inverse kinematics** to produce grasping configuration
  - Trivial in 2D, more interesting in general (7 DOF arm)

```
(:stream sample-ik
  :inputs (?b ?p)
  :domain (Pose ?b ?p)
  :outputs (?q)
  :certified (and (Conf ?q) (Kin ?b ?q ?p)))
```

```python
def sample_ik(b, p):
    q = p + GRASP
    yield (q,)
```

# 2D Pick-and-Place Streams



- "Sample" (via an RRT) **multi-waypoint trajectories**
- Include **joint limits & fixed obstacle collisions**, but not movable object collisions

```
(:stream sample-motion
   :inputs (?q1 ?q2)
   :domain (and (Conf ?q1) (Conf ?q2))
   :outputs (?t)
   :certified (and (Traj ?t) (Motion ?q1 ?t ?q2)))
```

```python
def sample_motion(q1, q2):
 t = rrt(q1, q2)
 if t is not None:
     yield (t,)
```

# 2D Pick-and-Place Collisions



- Need to ensure place actions do not cause **collisions**
- Add parameters for the pose of each block - **bad!**
- Use a **derived predicate** for whether currently **unsafe**
  - Decomposes collision checking into a logical **AND**

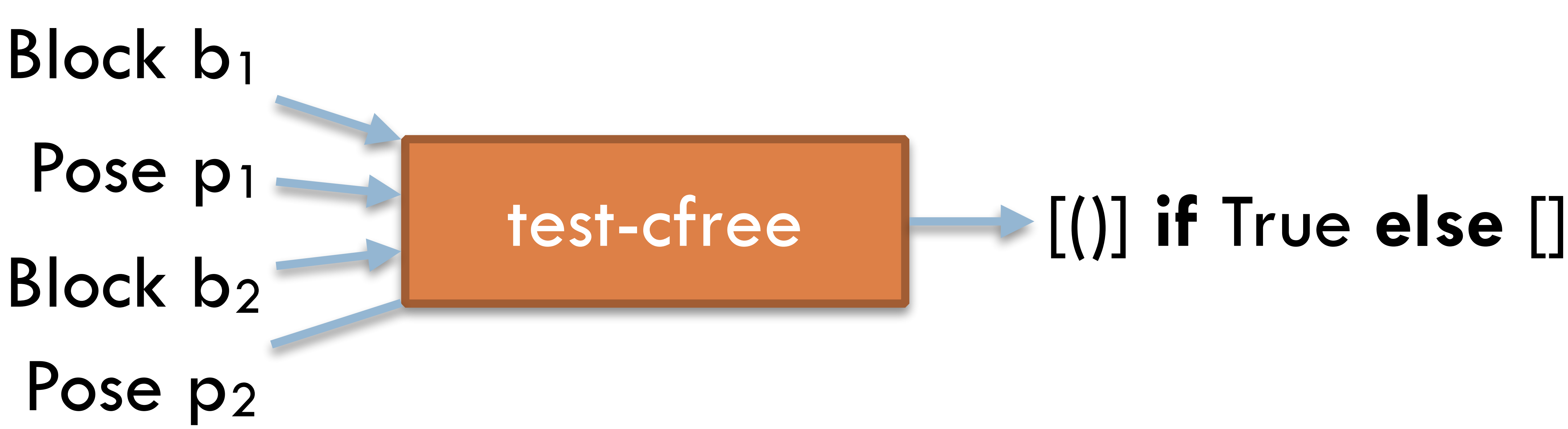
```
(:action place
   :parameters (?b ?p ?q)
   :precondition (and (Kin ?b ?q ?p)
                      (AtConf ?q) (Holding ?b)
                      (not (UnsafePose ?b ?p)))
   :effect (and (AtPose ?b ?p) (HandEmpty) (not (Holding ?b))))

(:derived (UnsafePose ?b1 ?p1)
   (exists (?b2 ?p2) (and (Pose ?b1 ?p1) (Pose ?b2 ?p2)
                          (AtPose ?b2 ?p2)
                          (not (CFree ?b1 ?p1 ?b2 ?p2)))))
```

# 2D Pick-and-Place Collisions



```
(:stream test_cfree
  :inputs (?b1 ?p1 ?b2 ?p2)
  :domain (and (Pose ?b1 ?p1) (Pose ?b2 ?p2))
  :outputs ()
  :certified (CFree ?b1 ?p1 ?b2 ?p2))
```

```python
def test_cfree(b1, p1, b2, p2):
    x1, y1 = p1
    x2, y2 = p2
    w1 = BLOCKS[b1].width
    w2 = BLOCKS[b2].width
    if (w1 + w2)/2 <= abs(x2 - x1):
        yield tuple()
```

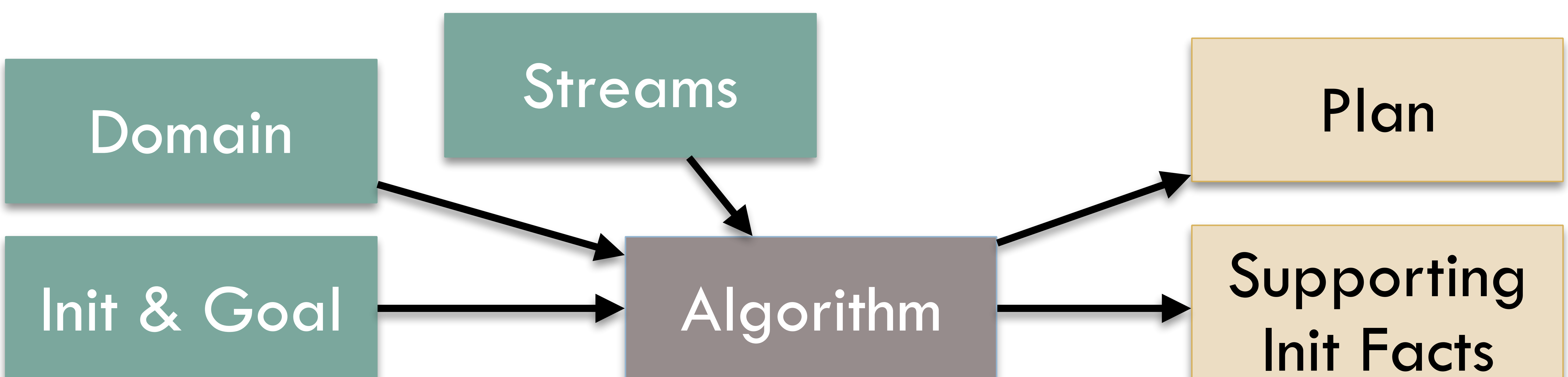


# PDDLStream Algorithms

# PDDLStream Specification



- **Domain dynamics** - *domain.pddl*
  - Specifies actions and derived predicates
- **Stream properties** - *stream.pddl*
  - Declares input/output arity as well as certified facts
- **Problem** and **stream implementation** - *problem.py*
  - Initial state a set of facts & goal formula
  - Stream implementation in Python



# Two PDDLStream Algorithms

- Each algorithm repeats:
  1. **Search** a finite PDDL problem for plan
  2. **Modify** the PDDL problem depending on the plan

- Search implemented using **blackbox algorithms**
  - Breadth-First Search (BFS)
  - **Off-the-shelf AI planner** - FastDownard
    - Exploits factoring in its search heuristics (e.g. $h_{FF}$)
    - http://www.fast-downward.org/

# Incremental Algorithm

- Incrementally construct all possible initial facts
- Periodically check if admits a solution
- Alternation between **sampling and searching** is similar in spirit to a **Probabilistic Roadmap** (PRM)

- Repeat:

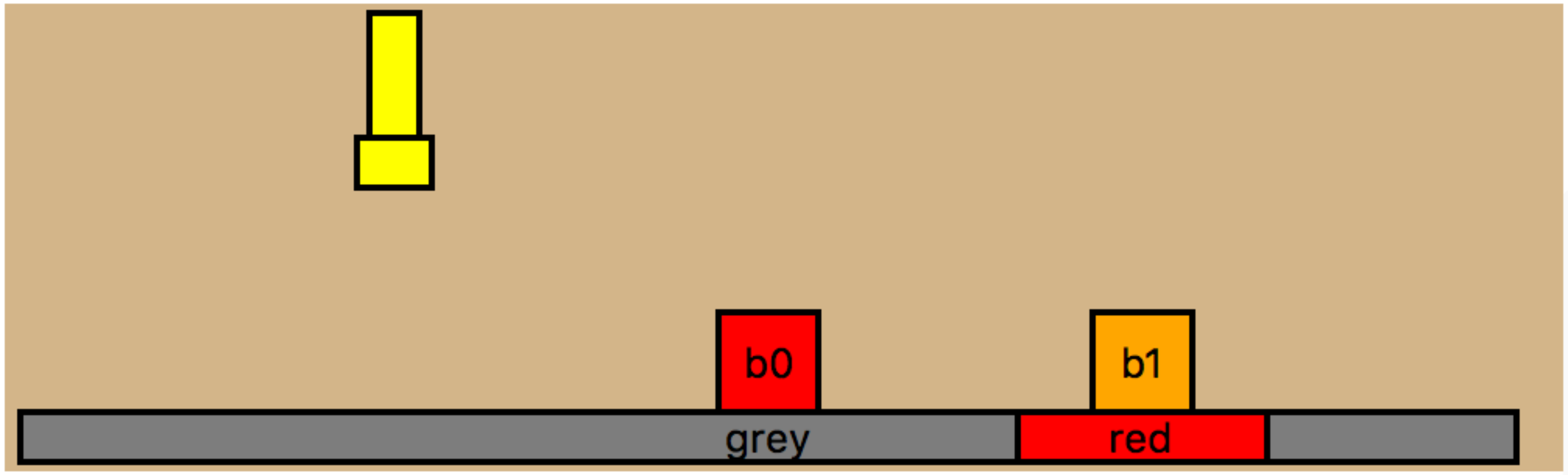
1. **Compose** and **evaluate** a finite number of streams to unveil more facts in the initial state
2. **Search** the current PDDL problem for plan
3. **Terminate** when a plan is found

# Incremental Example: Iteration 1



**Stream evaluations:**

1. **s-motion:**([-7.5  5. ], [-7.5  5. ])->[([[-7.5  5. ], [-7.5  5. ], [-7.5  5. ], [-7.5  5. ]])]
2. **s-ik:**(b0, [0. 0.])->[([0.  2.5])]
3. **t-cfree:**(b0, [0. 0.], b0, [0. 0.])->[] **(failed to produce an output)**
4. **s-ik:**(b1, [7.5 0. ])->[([7.5 2.5])]
5. **t-cfree:**(b1, [7.5 0. ], b0, [0. 0.])->[()]
6. **t-cfree:**(b0, [0. 0.], b1, [7.5 0. ])->[()]
7. **t-cfree:**(b1, [7.5 0. ], b1, [7.5 0. ])->[] **(failed to produce an output)**
8. **t-region:**(b0, [0. 0.], grey)->[()]
9. **t-region:**(b1, [7.5 0. ], grey)->[()]
10. **t-region:**(b0, [0. 0.], red)->[] **(failed to produce an output)**
11. **s-region:**(b0, red)->[([7.65 0.  ])]
12. **s-region:**(b1, red)->[([8.15 0.  ])]
13. **s-region:**(b0, grey)->[([2.88 0.  ])]
14. **s-region:**(b1, grey)->[([1.26 0.  ])]



# Incremental Example: Iteration 2



**Stream evaluations**

1. **s-motion:**([0.  2.5], [-7.5  5. ])->[([[0.  2.5], [0. 5.], [-7.5  5. ], [-7.5  5. ]])]
2. **s-motion:**([-7.5  5. ], [0.  2.5])->[([[-7.5  5. ], [-7.5  5. ], [0. 5.], [0.  2.5]])]
3. **s-motion:**([0.  2.5], [0.  2.5])->[([[0.  2.5], [0. 5.], [0. 5.], [0.  2.5]])]
4. **s-motion:**([7.5 2.5], [-7.5  5. ])->[([[7.5 2.5], [7.5 5. ], [-7.5  5. ], [-7.5  5. ]])]
5. **s-motion:**([7.5 2.5], [0.  2.5])->[([[7.5 2.5], [7.5 5. ], [0. 5.], [0.  2.5]])]
6. **s-motion:**([-7.5  5. ], [7.5 2.5])->[([[-7.5  5. ], [-7.5  5. ], [7.5 5. ], [7.5 2.5]])]
7. **s-motion:**([0.  2.5], [7.5 2.5])->[([[0.  2.5], [0. 5.], [7.5 5. ], [7.5 2.5]])]
8. **s-motion:**([7.5 2.5], [7.5 2.5])->[([[7.5 2.5], [7.5 5. ], [7.5 5. ], [7.5 2.5]])]
9. **s-ik:**(b0, [7.65 0.  ])->[([7.65 2.5 ])]
10. **t-cfree:**(b0, [7.65 0.  ], b0, [0. 0.])->[()]
11. **t-cfree:**(b0, [7.65 0.  ], b1, [7.5 0. ])->[] **(failed to produce an output)**
12. **t-cfree:**(b0, [0. 0.], b0, [7.65 0.  ])->[()]
13. **t-cfree:**(b1, [7.5 0. ], b0, [7.65 0.  ])->[] **(failed to produce an output)**
14. **t-cfree:**(b0, [7.65 0.  ], b0, [7.65 0.  ])->[] **(failed to produce an output)**
15. **t-region:**(b0, [7.65 0.  ], grey)->[()]
16. **t-region:**(b0, [7.65 0.  ], red)->[()]
17. <u>**s-region:**(b0, red)->[([7.27 0.  ])] (second generator output)</u>
18. **s-ik:**(b1, [8.15 0.  ])->[([8.15 2.5 ])]

…

54. <u>**s-region:**(b1, grey)->[([10.97  0.  ])] (second generator output)</u>

# Incremental Example: Iterations 3-4



**Iteration 3** - 118 stream evaluations
**Iteration 4** - 182 stream evaluations

**Plan:**
1) **move** [-7.5  5. ] [[-7.5  5. ], [-7.5  5. ], [7.5 5. ], [7.5 2.5]] [7.5 2.5]
2) **pick** b1 [7.5 0. ] [7.5 2.5]
3) **move** [7.5 2.5] [[7.5 2.5], [7.5 5. ], [10.97  5.  ], [10.97  2.5 ]] [10.97  2.5 ]
4) **place** b1 [10.97  0.  ] [10.97  2.5 ]
5) **move** [10.97  2.5 ] [[10.97  2.5 ], [10.97  5.  ], [0. 5.], [0.  2.5]] [0.  2.5]
6) **pick** b0 [0. 0.] [0.  2.5]
7) **move** [0.  2.5] [[0.  2.5], [0. 5.], [7.65 5.  ], [7.65 2.5 ]] [7.65 2.5 ]
8) **place** b0 [7.65 0.  ] [7.65 2.5 ]

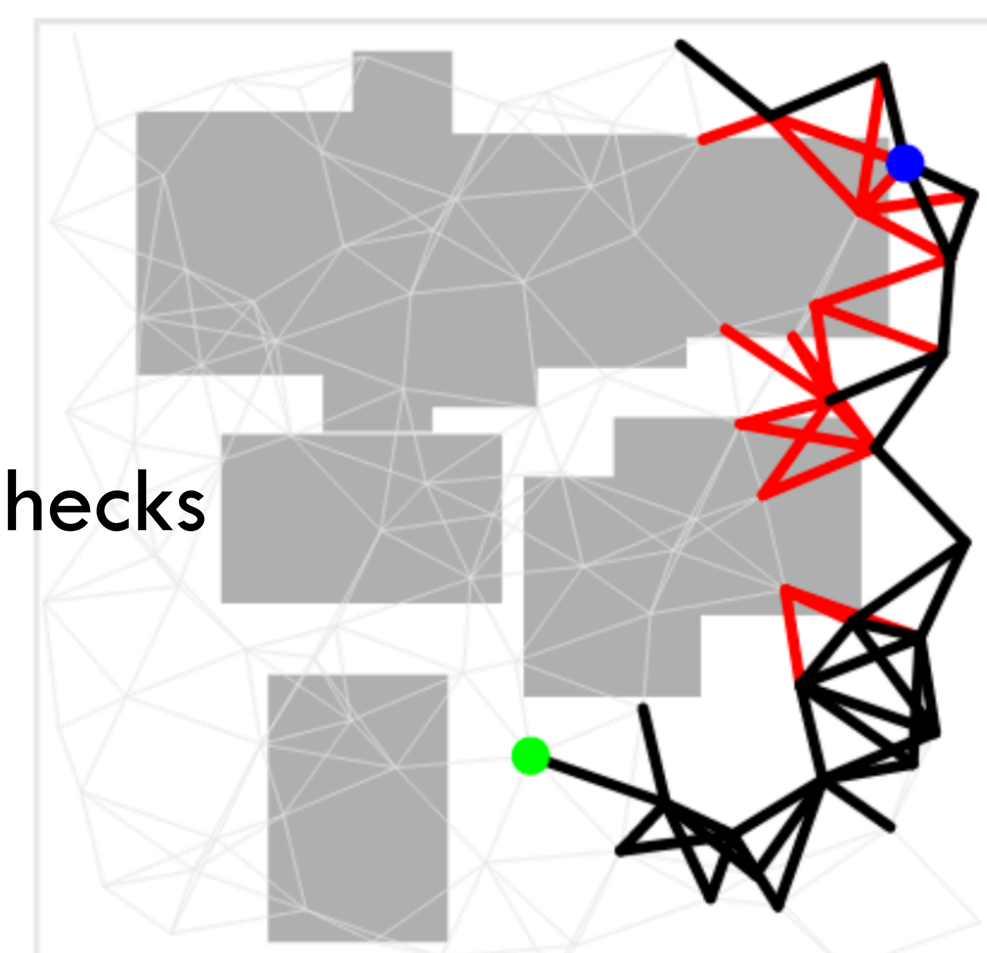
- **Drawback** - many unnecessary samples produced
  - **Computationally expensive** to generate
  - Induce **large discrete planning problems**
- Motivates designing more intelligent algorithms

# Lazy Probabilistic Roadmap (PRM)

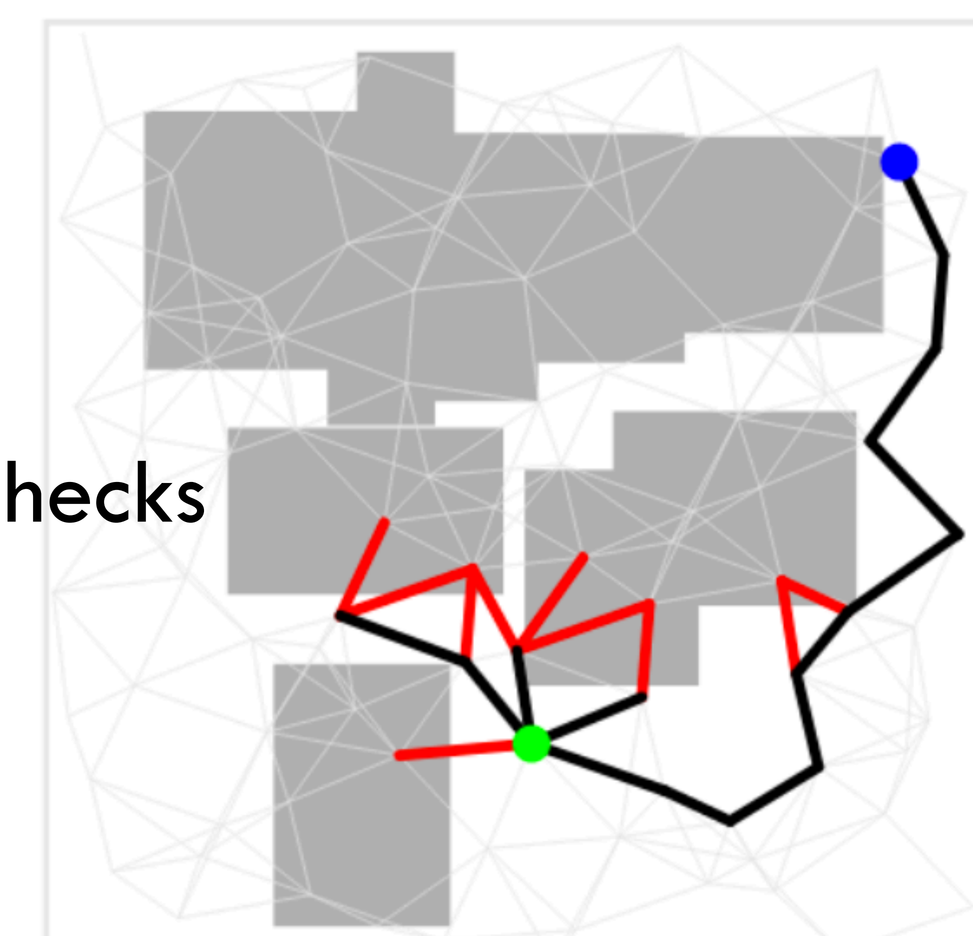
- Motivated by **expensive collision checks**
  - *[Bohlin & Kavraki, Dellin & Srinivasa]*
- **Defers collision checking** until a path is found

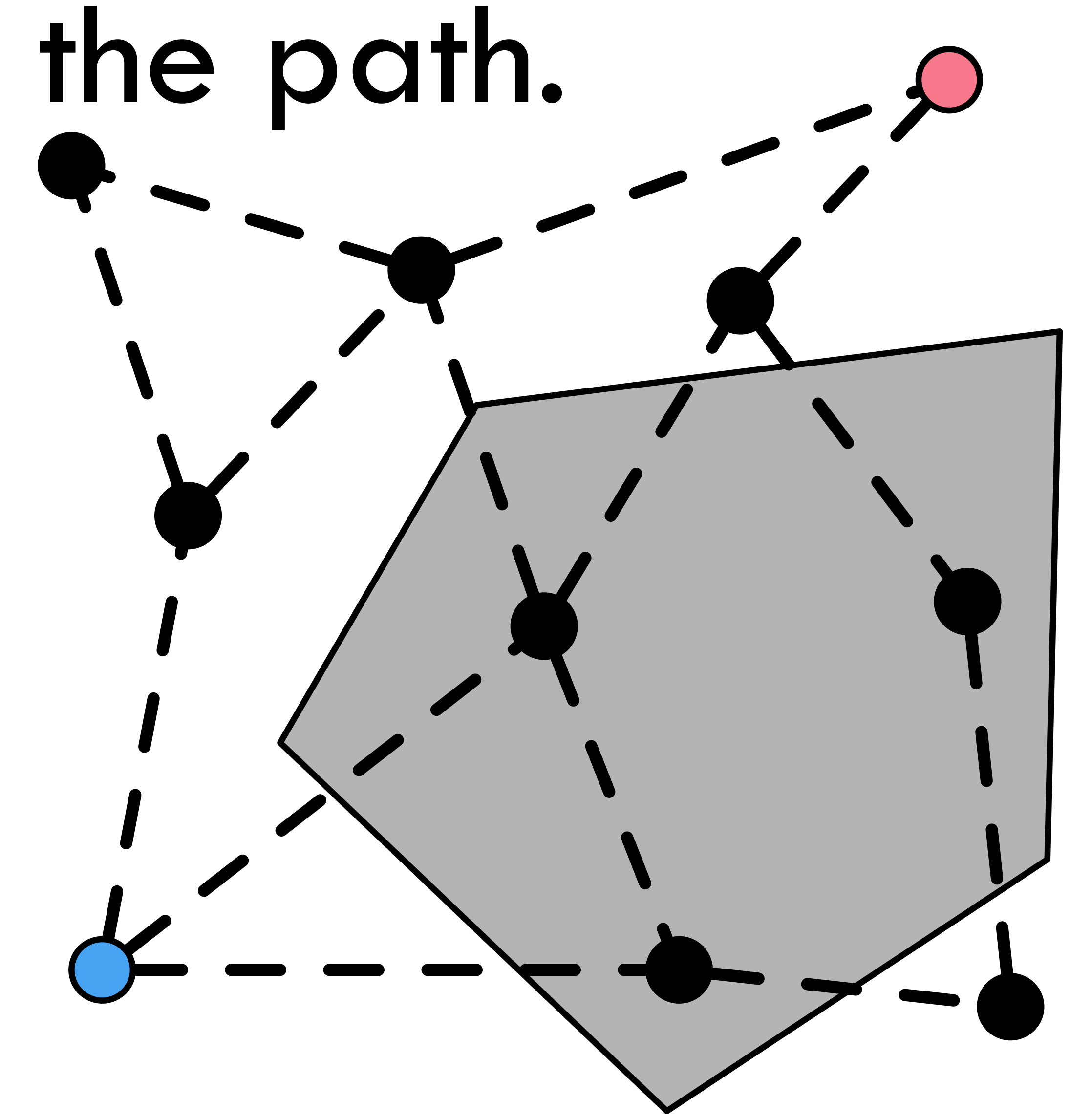
1. Find a path using the **unchecked and safe** edges
2. Check collisions for edges **only along the path**
3. If collision, **goto** 1. Otherwise, **return** the path.

77 checks



23 checks



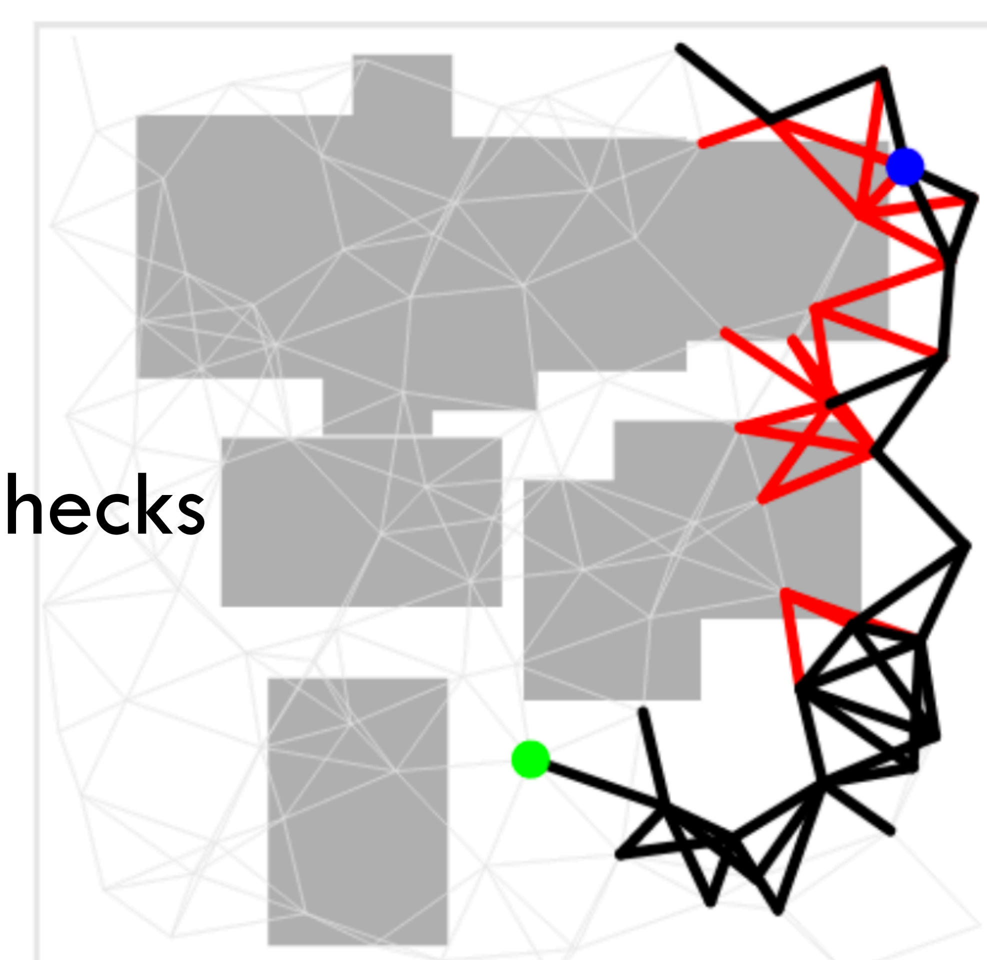


# Lazy Probabilistic Roadmap (PRM)

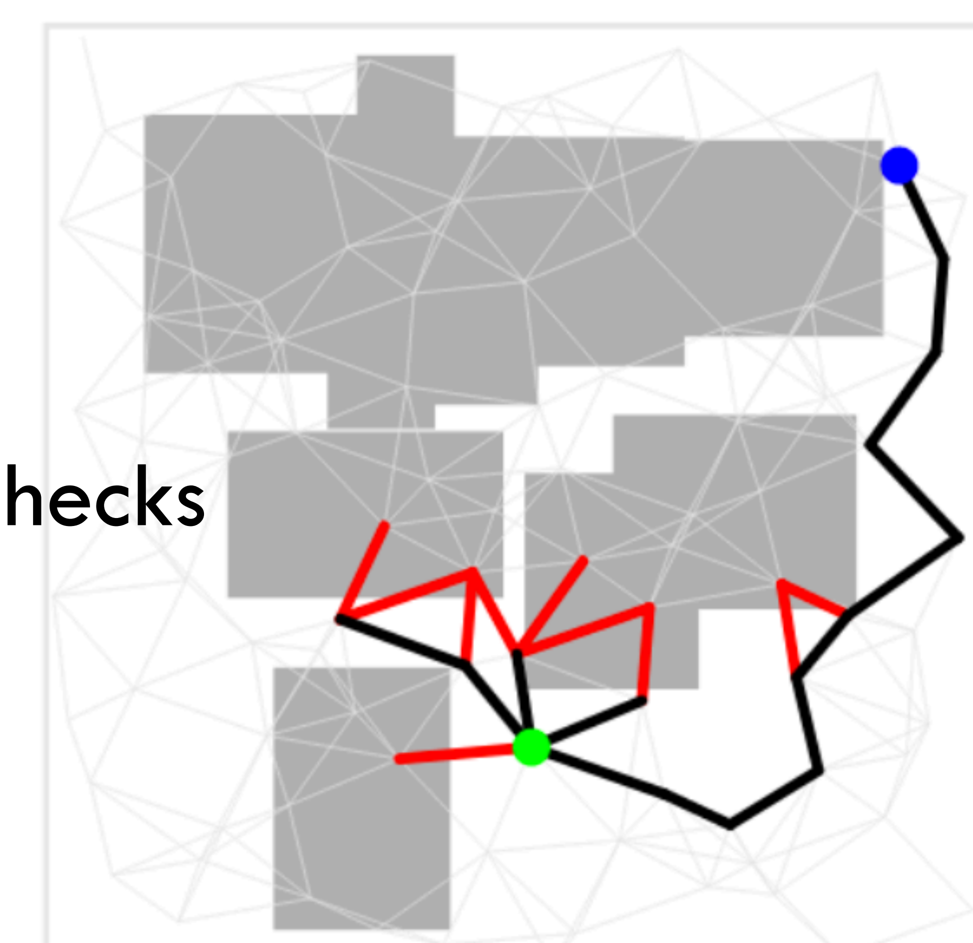
- Motivated by **expensive collision checks**
  - *[Bohlin & Kavraki, Dellin & Srinivasa]*
- **Defers collision checking** until a path is found

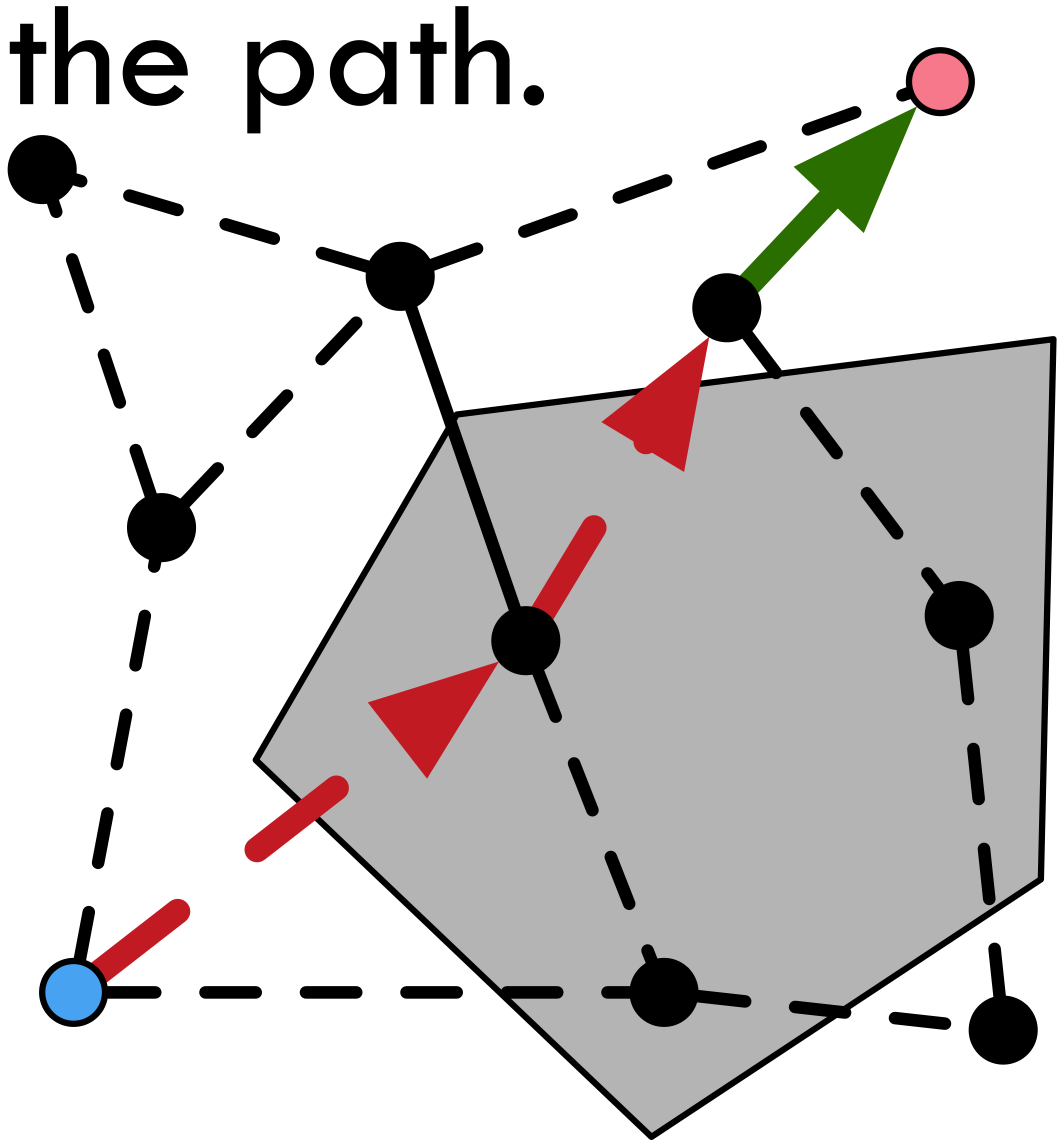
1. Find a path using the **unchecked and safe** edges
2. Check collisions for edges **only along the path**
3. If collision, **goto** 1. Otherwise, **return** the path.

77 checks



23 checks



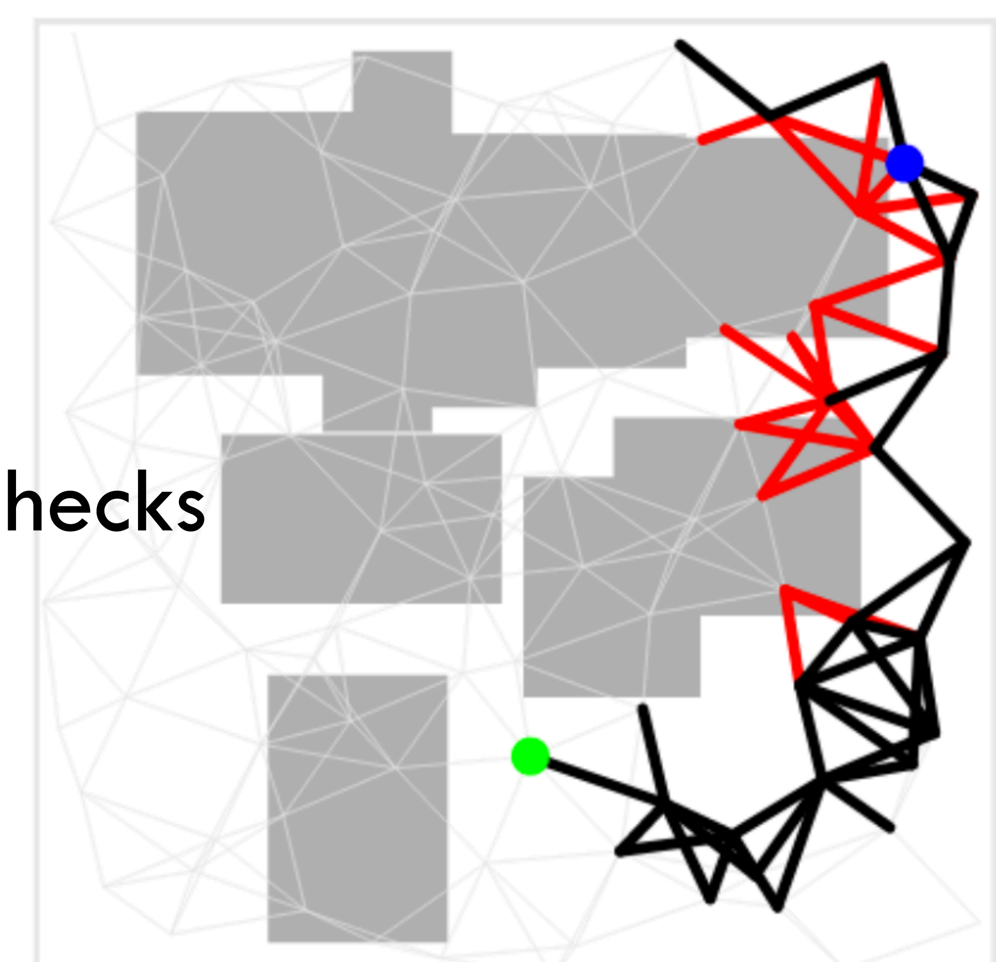


# Lazy Probabilistic Roadmap (PRM)

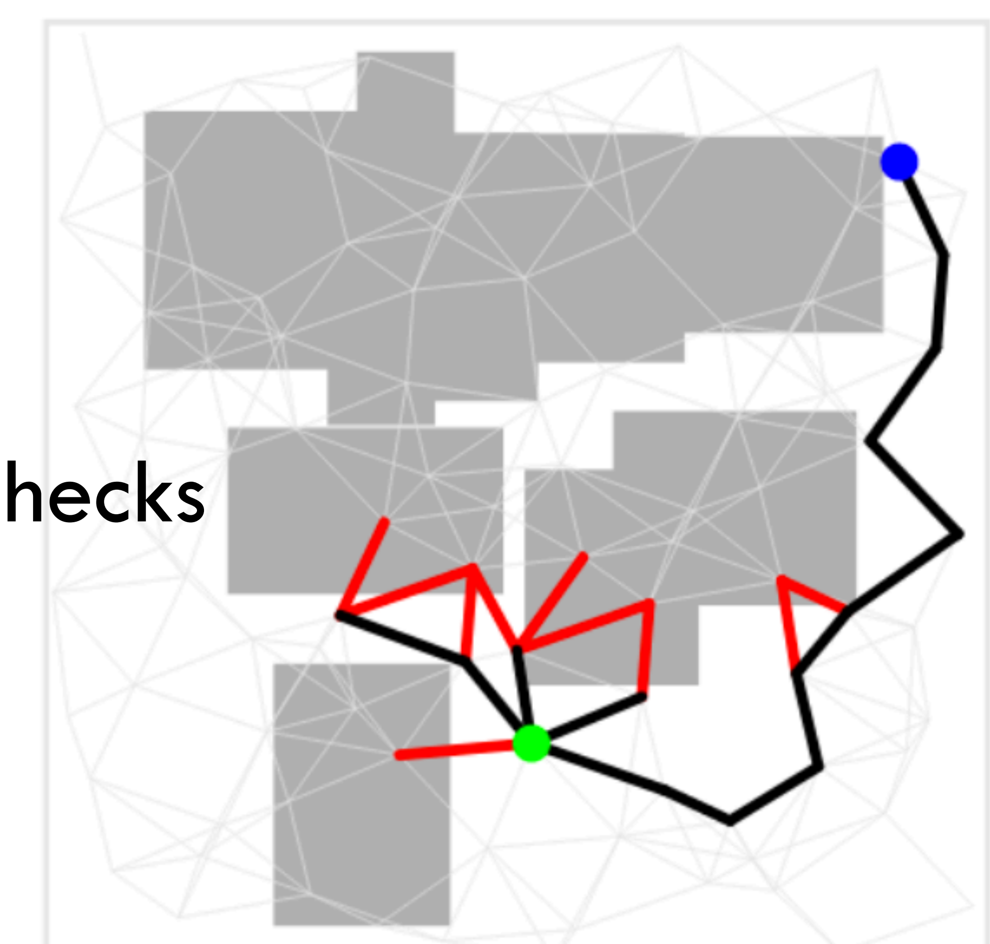
- Motivated by **expensive collision checks**
  - *[Bohlin & Kavraki, Dellin & Srinivasa]*
- **Defers collision checking** until a path is found

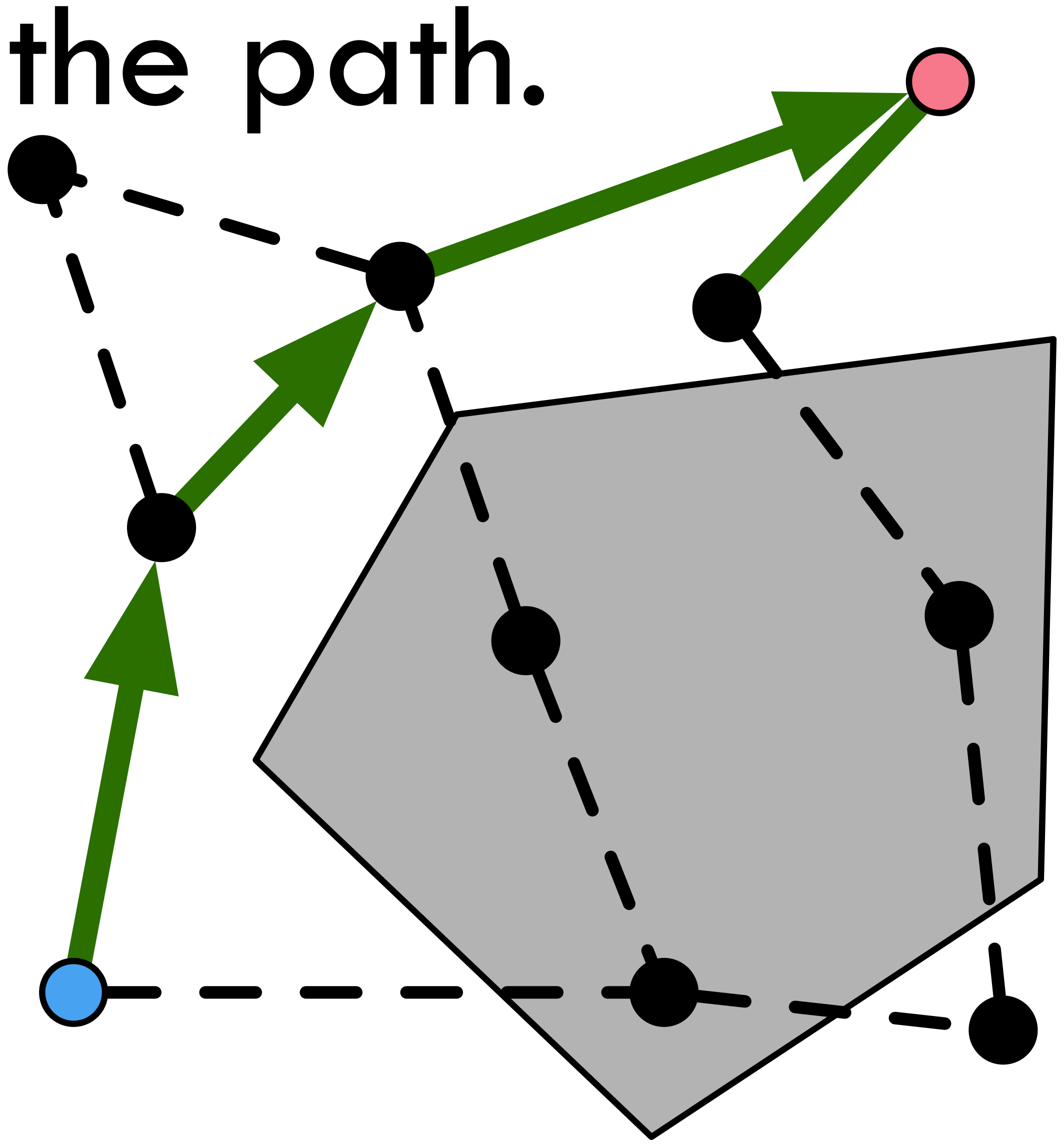
1. Find a path using the **unchecked and safe** edges
2. Check collisions for edges **only along the path**
3. If collision, **goto** 1. Otherwise, **return** the path.

77 checks



23 checks





# Optimistic Stream Outputs



- Many TAMP streams are exceptionally **expensive**
  - Inverse kinematics, motion planning, …
- **Generalize lazy collision checking** idea to streams
- Inductively create **unique placeholder** output objects for each stream instance (has **#** as its prefix)

- **Optimistic evaluations:**

1. [**s-region:**(b0, red)->(**#p0**), …
2. [**s-ik:**(b0, [0. 0.])->(**#q0**), s-ik:(b0, **#p0**)->(**#q2**), **t-cfree:** (b0, **#p0**, b1, [7.5 0. ])->() …
3. [**s-motion:**([-7.5  5. ], **#q0**)->(**#t2**), **s-motion:**(**#q0**, **#q2**)->(**#t13**), …]

# Focused Algorithm



- Plan using optimistic stream outputs **before** evaluating stream instances
- **Recover** set of possibly helpful stream instances from identified plans

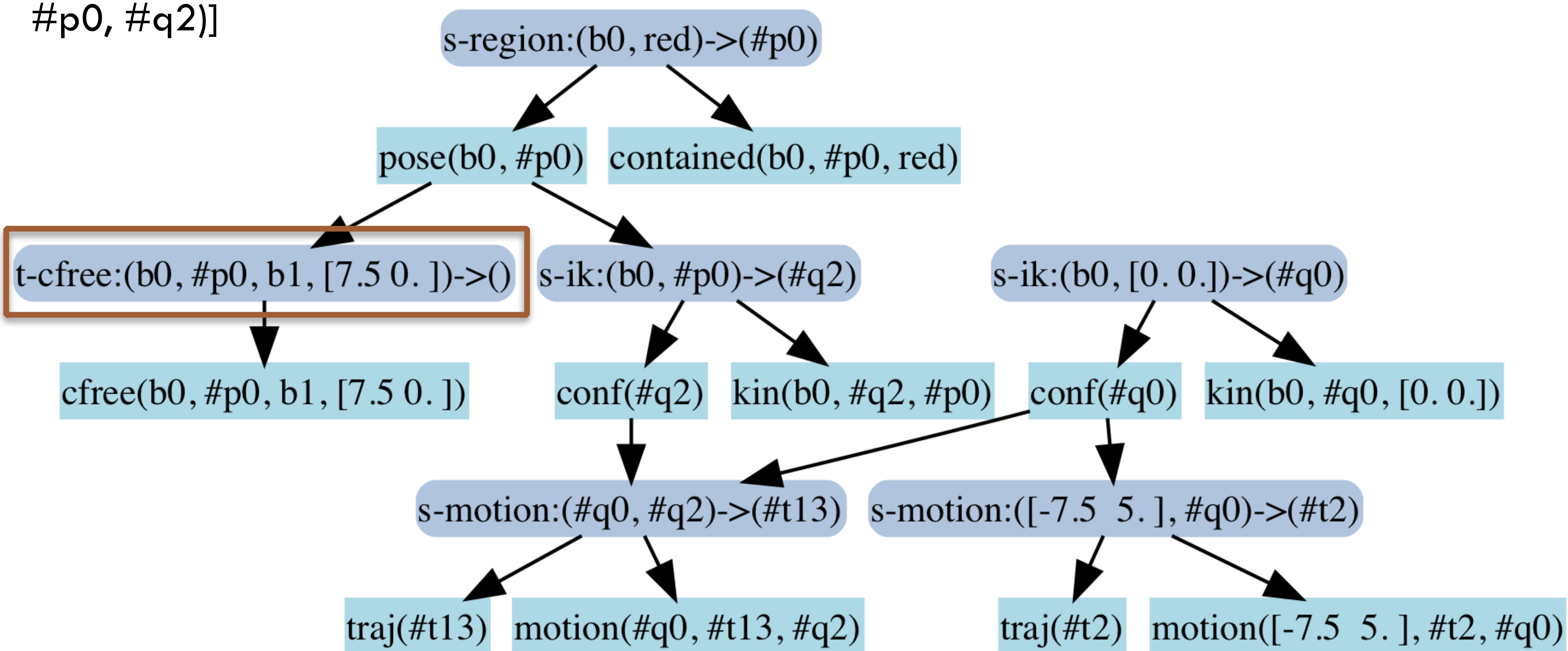
- Repeat:

1. Construct current **optimistic** objects & facts
2. **Search** with **real & optimistic** objects
3. If **only real objects** used, **return plan**
4. **Evaluate** stream instances supporting plan
5. **Disable** evaluated stream instances

# Focused Example: Iteration 1



**Plan:** [**move**([-7.5  5. ], #t2, #q0), **pick**(b0, [0. 0.], #q0), **move**(#q0, #t13, #q2), **place**(b0, #p0, #q2)]




**Stream evaluations:**

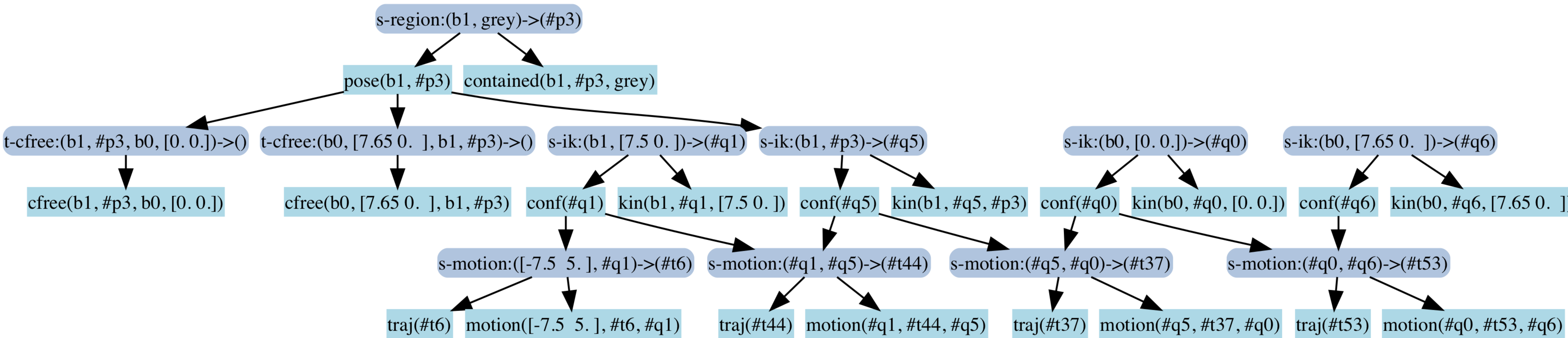
1. **s-region:**(b0, red)->[([7.65 0.  ])]
2. **t-cfree:**(b0, [7.65 0.  ], b1, [7.5 0. ])->[]  **(failed to produce an output)**

These stream instances are **removed** from subsequent searches

# Focused Example: Iteration 2



**Plan**: [**move**([-7.5  5. ], #t6, #q1), **pick**(b1, [7.5 0. ], #q1), **move**(#q1, #t44, #q5), **place**(b1, #p3, #q5), **move**(#q5, #t37, #q0), **pick**(b0, [0. 0.], #q0), **move**(#q0, #t53, #q6), **place**(b0, [7.65 0.  ], #q6)]




**Stream evaluations:**
1. **s-region:**(b1, grey)->[([2.88 0.  ])]
2. **t-cfree:**(b1, [2.88 0.  ], b0, [0. 0.])->[()], **t-cfree:**(b0, [7.65 0.  ], b1, [2.88 0.  ])->[()]
3. **s-ik:**(b0, [7.65 0.  ])->[([7.65 2.5 ])], **s-ik:**(b1, [7.5 0. ])->[([7.5 2.5])]
4. **s-ik:**(b1, [2.88 0.  ])->[([2.88 2.5 ])], **s-ik:**(b0, [0. 0.])->[([0.  2.5])]
5. **s-motion:**([7.5 2.5], [2.88 2.5 ])->[([[7.5 2.5], [7.5 5. ], [2.88 5.  ], [2.88 2.5 ]])]
6. **s-motion:**([-7.5  5. ], [7.5 2.5])->[([[-7.5  5. ], [-7.5  5. ], [7.5 5. ], [7.5 2.5]])]
7. **s-motion:**([0.  2.5], [7.65 2.5 ])->[([[0.  2.5], [0. 5.], [7.65 5.  ], [7.65 2.5 ]])]
8. **s-motion:**([2.88 2.5 ], [0.  2.5])->[([[2.88 2.5 ], [2.88 5.  ], [0. 5.], [0.  2.5]])]

# Shared Optimistic Objects



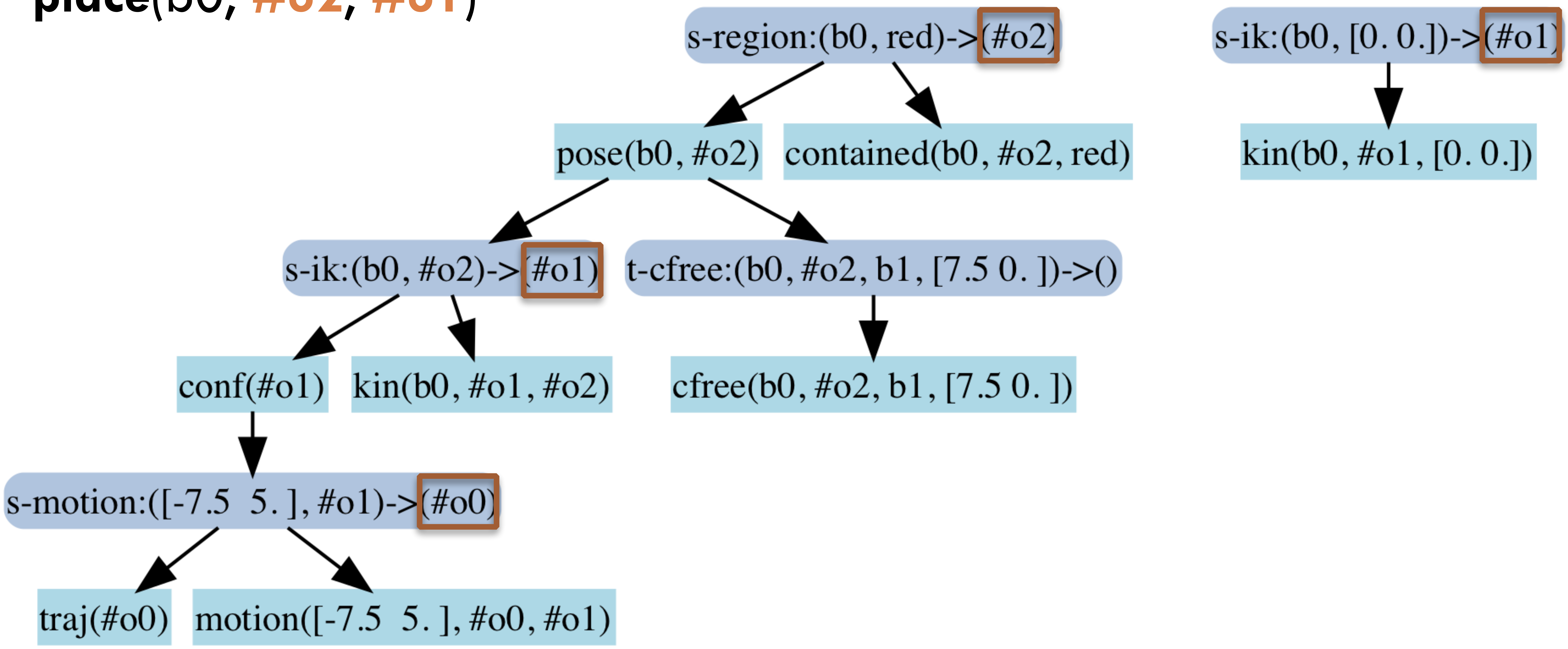
- Focused still makes a **large number** of optimistic objects
  - Induce **large discrete planning problems**
- While many objects could be created, **only a few will be used** on a solution
- Let optimistic objects be the output of **several streams**
  - Typically **overly optimistic**, resolve through differentiation and additional search

- For example: single object **#o1** for each **s-ik** input
  - **s-ik:**(b0, #o2)->(**#o1**)
  - **s-ik:**(b0, [0. 0.])->**(#o1)**

# Focused Refinement: Step 1



**Plan:** [**move**([-7.5  5. ], **#o0**, **#o1**), **pick**(b0, [0. 0.], **#o1**), **place**(b0, **#o2**, **#o1**)]



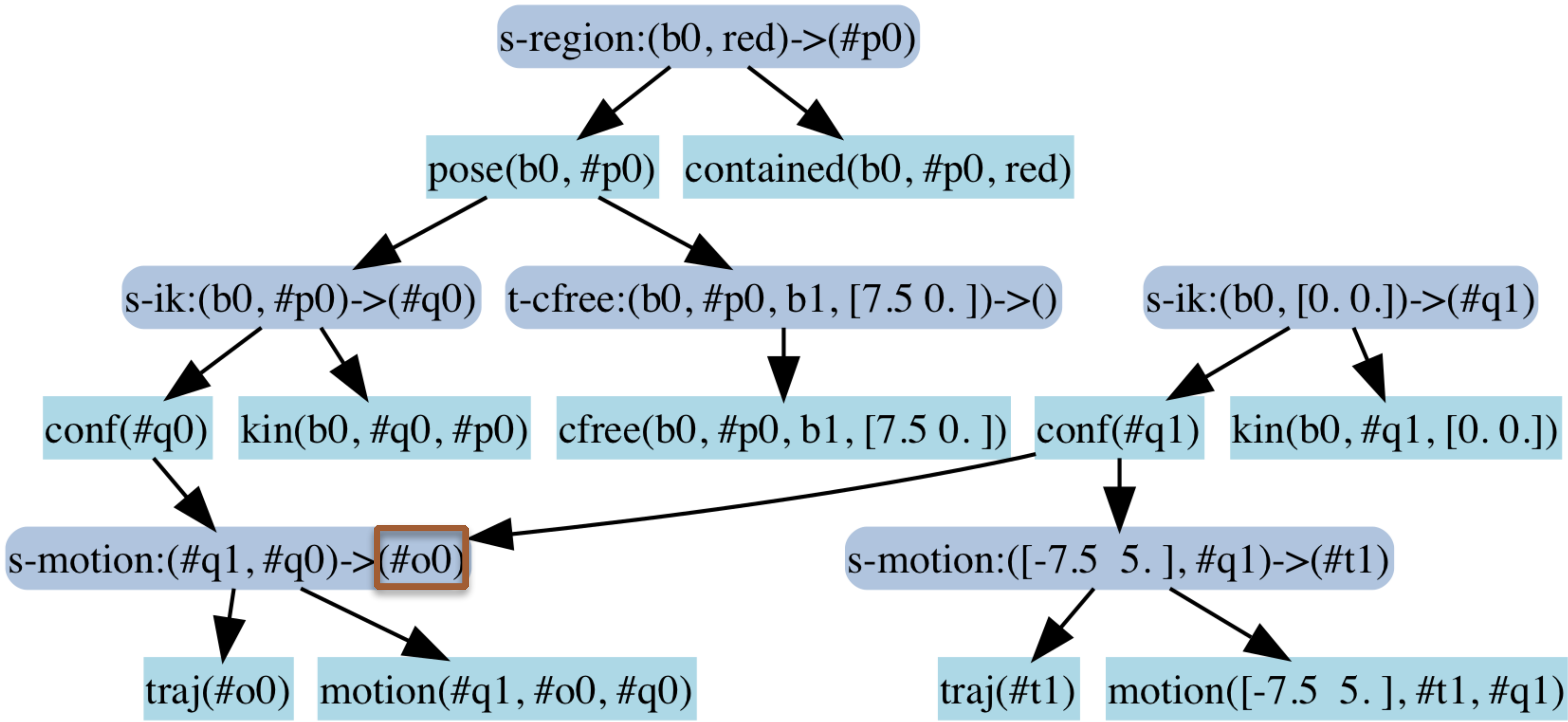

**Unique optimistic objects** are created for:

**s-region:**(b0, red)->(**#o2**), **s-ik:**(b0, [0. 0.])->(**#o1**)

# Focused Refinement: Step 2



**Plan:** [**move**([-7.5  5. ], #t1, #q1), **pick**(b0, [0. 0.], #q1), **move**(#q1, **#o0**, #q0), **place**(b0, #p0, #q0)]



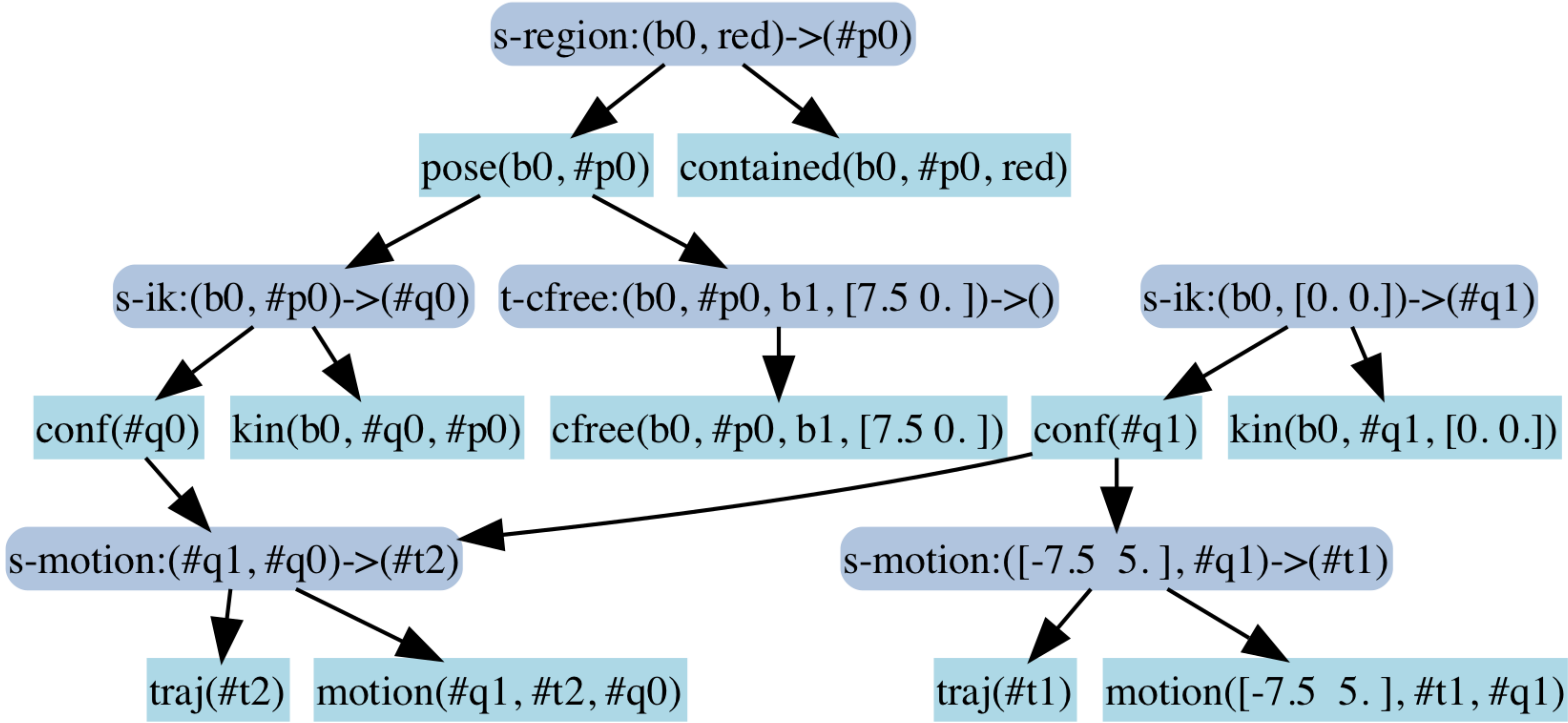

**Unique optimistic objects** are created for:

**s-motion:**(#q1, #q0)->(**#o0**)

# Focused Refinement: Step 3



**Plan:** [**move**([-7.5  5. ], #t1, #q1), **pick**(b0, [0. 0.], #q1), **move**(#q1, #t2, #q0), **place**(b0, #p0, #q0)]




**No more** shared optimistic objects!

Focus proceeds as previously described

# Theoretical Analysis



- PDDLStream plan existence is **undecidable** but **semi-decidable**
  - Can encode **halting problem** within a stream
- Incremental and focused algorithms are **semi-complete**
  - Find a plan if streams will eventually produce one

- For TAMP, can lead to **resolution complete** and **probabilistically complete** algorithms
  - If robustly feasible, will find a plan with **probability 1**
  - Streams are part of the **algorithm instead of input**

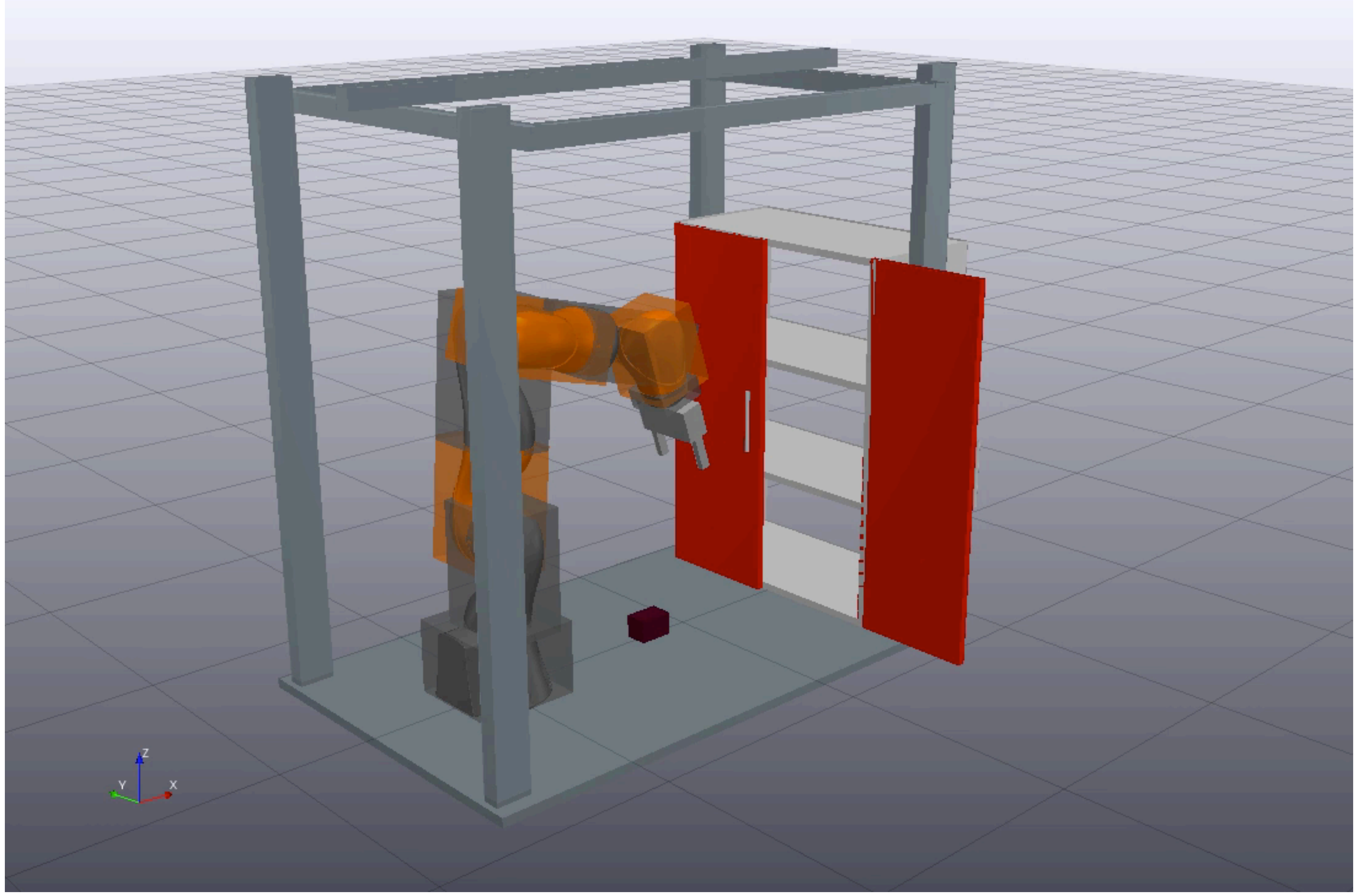
# Extensions

# PDDLStream + Drake



- **Goal**: brick on the **second cupboard shelf** and robot + doors at **initial joint positions**
- Must **open** the left door to safely place and then **close**
- No need to manipulate the right door

1. Visual object **detections**
2. Point cloud **pose estimation**
3. **Solve** a TAMP problem to obtain joint trajectories
4. Convert to iiwa splines & gripper set points
5. **Execute** using controllers

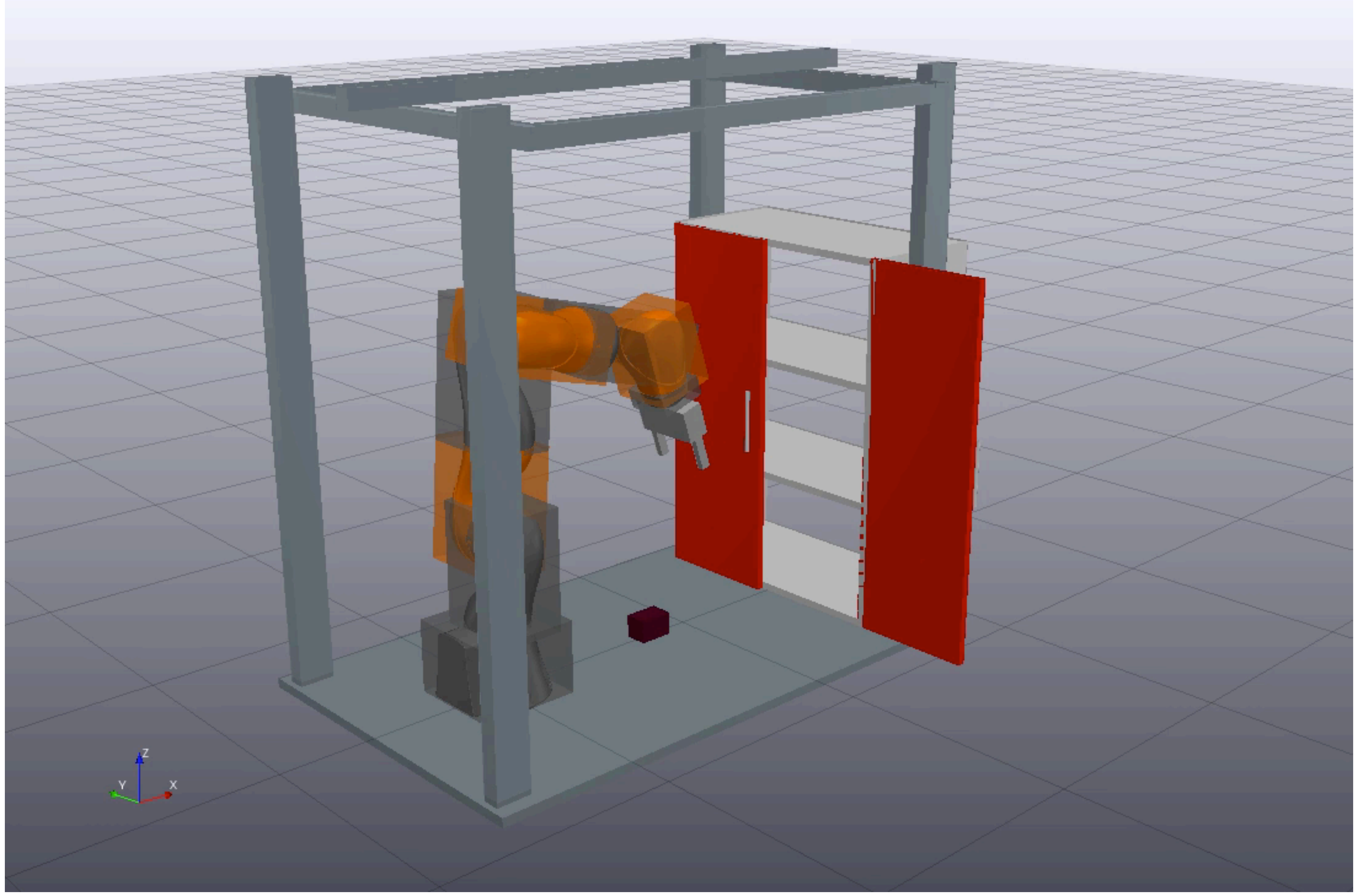


# PDDLStream + Drake



- **Goal**: brick on the **second cupboard shelf** and robot + doors at **initial joint positions**
- Must **open** the left door to safely place and then **close**
- No need to manipulate the right door

1. Visual object **detections**
2. Point cloud **pose estimation**
3. **Solve** a TAMP problem to obtain joint trajectories
4. Convert to iiwa splines & gripper set points
5. **Execute** using controllers



# PDDLStream + Drake Actions



- Problem formulation is **very similar** to the 2D pick-and-place example
  - Configurations & poses simply have **more DOFs**

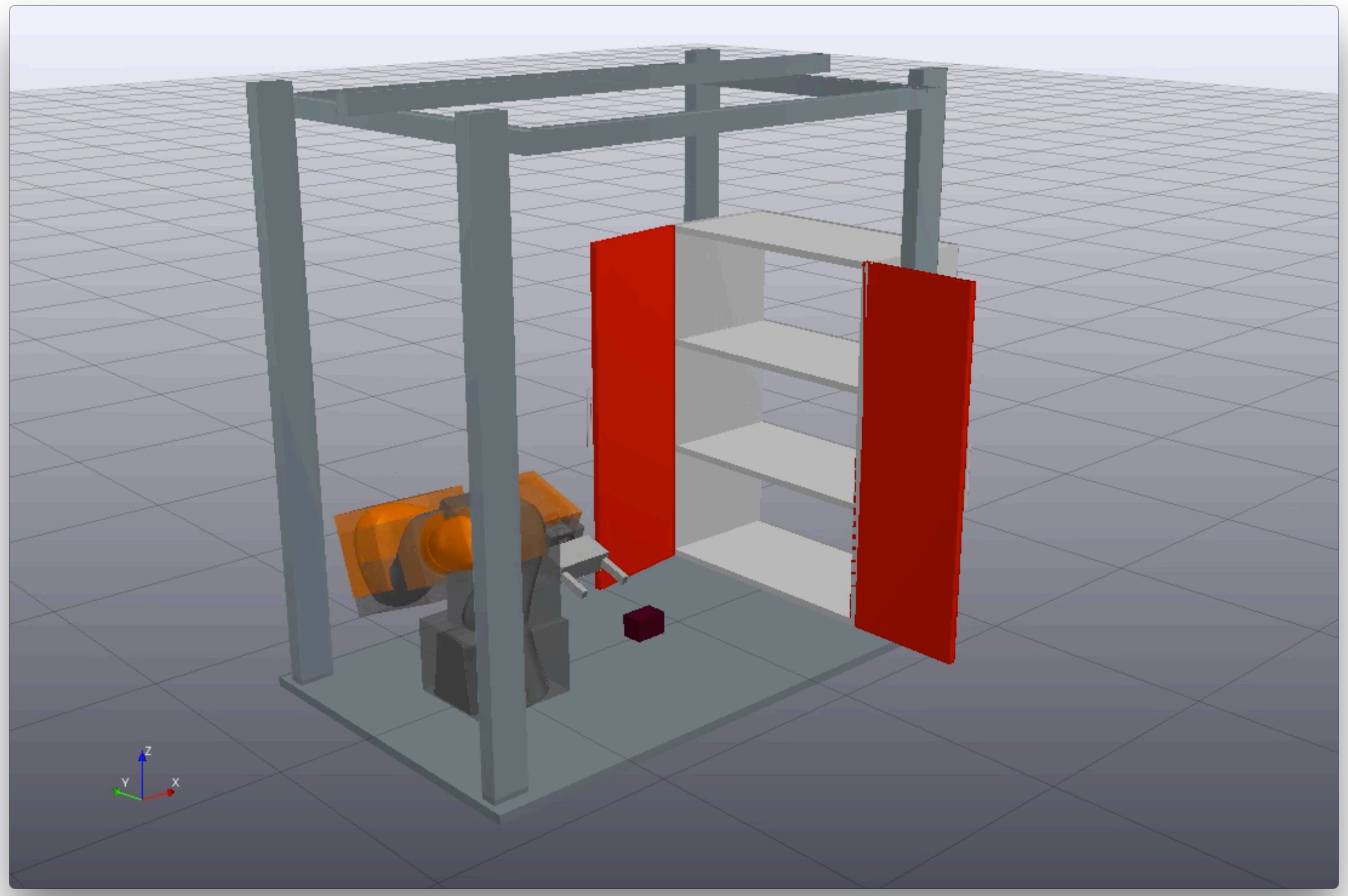
```
(:action place
  :parameters (?r ?o ?p ?g ?q ?t)
  :precondition (and (Kin ?r ?o ?p ?g ?q ?t)
                     (AtGrasp ?r ?o ?g)
                     (AtConf ?r ?q)
                     (not (UnsafeTraj ?t)))
  :effect (and (AtPose ?o ?p)
               (HandEmpty ?r)
               (not (AtGrasp ?r ?o ?g))))

(:action pull
 … ; TODO: implementation
```

# PDDLStream + Drake Actions



- Problem formulation is **very similar** to the 2D pick-and-place example
  - Configurations & poses simply have **more DOFs**

```
(:action place
  :parameters (?r ?o ?p ?g ?q ?t)
  :precondition (and (Kin ?r ?o ?p ?g ?q ?t)
                     (AtGrasp ?r ?o ?g)
                     (AtConf ?r ?q)
                     (not (UnsafeTraj ?t)))
  :effect (and (AtPose ?o ?p)
               (HandEmpty ?r)
               (not (AtGrasp ?r ?o ?g))))

(:action pull
 … ; TODO: implementation
```

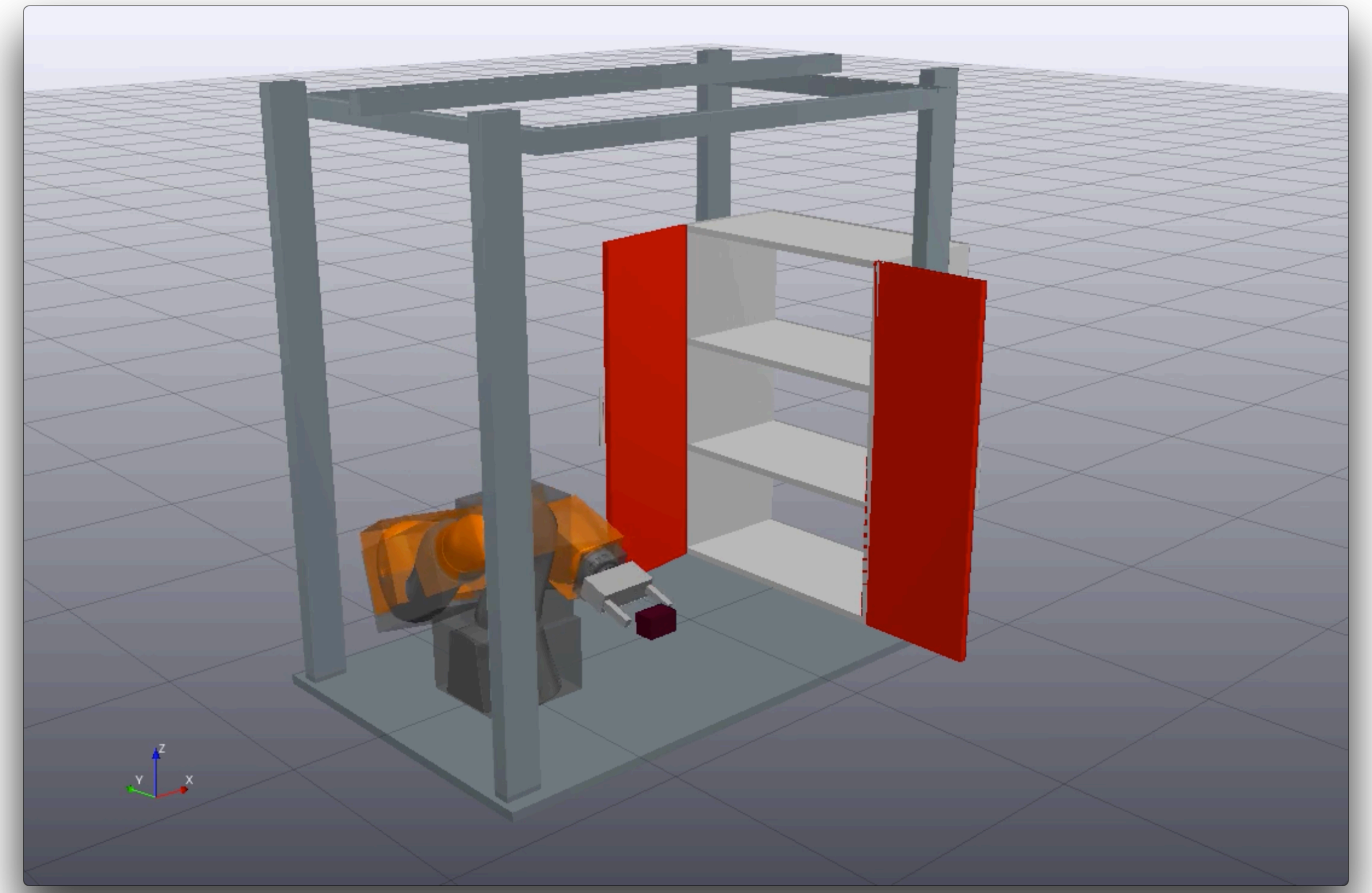


# PDDLStream + Drake Actions



- Problem formulation is **very similar** to the 2D pick-and-place example
  - Configurations & poses simply have **more DOFs**

```
(:action place
  :parameters (?r ?o ?p ?g ?q ?t)
  :precondition (and (Kin ?r ?o ?p ?g ?q ?t)
                     (AtGrasp ?r ?o ?g)
                     (AtConf ?r ?q)
                     (not (UnsafeTraj ?t)))
  :effect (and (AtPose ?o ?p)
               (HandEmpty ?r)
               (not (AtGrasp ?r ?o ?g))))

(:action pull
 … ; TODO: implementation
```

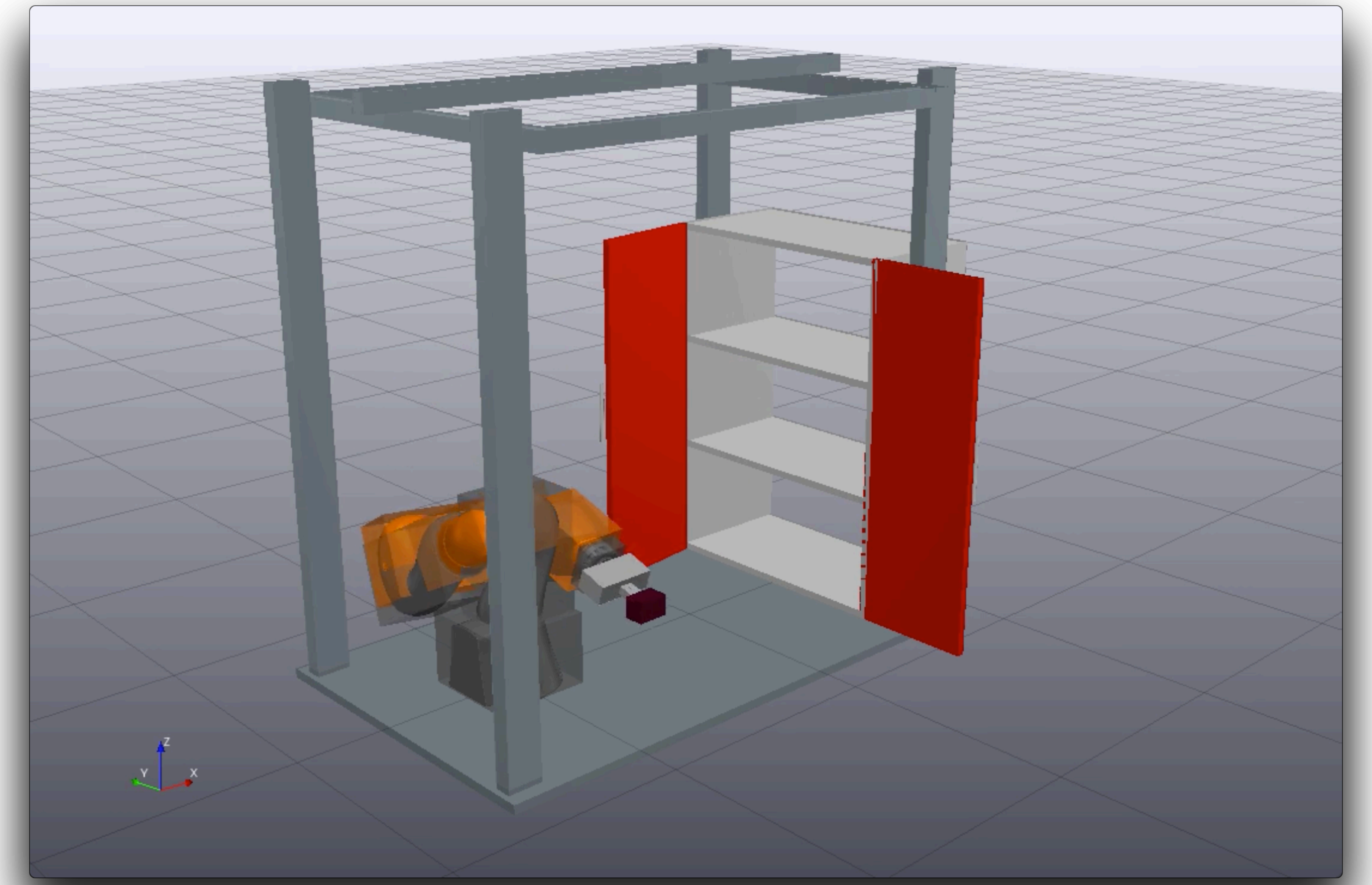


# PDDLStream + Drake Actions



- Problem formulation is **very similar** to the 2D pick-and-place example
  - Configurations & poses simply have **more DOFs**

```
(:action place
  :parameters (?r ?o ?p ?g ?q ?t)
  :precondition (and (Kin ?r ?o ?p ?g ?q ?t)
                     (AtGrasp ?r ?o ?g)
                     (AtConf ?r ?q)
                     (not (UnsafeTraj ?t)))
  :effect (and (AtPose ?o ?p)
               (HandEmpty ?r)
               (not (AtGrasp ?r ?o ?g))))

(:action pull
 … ; TODO: implementation
```

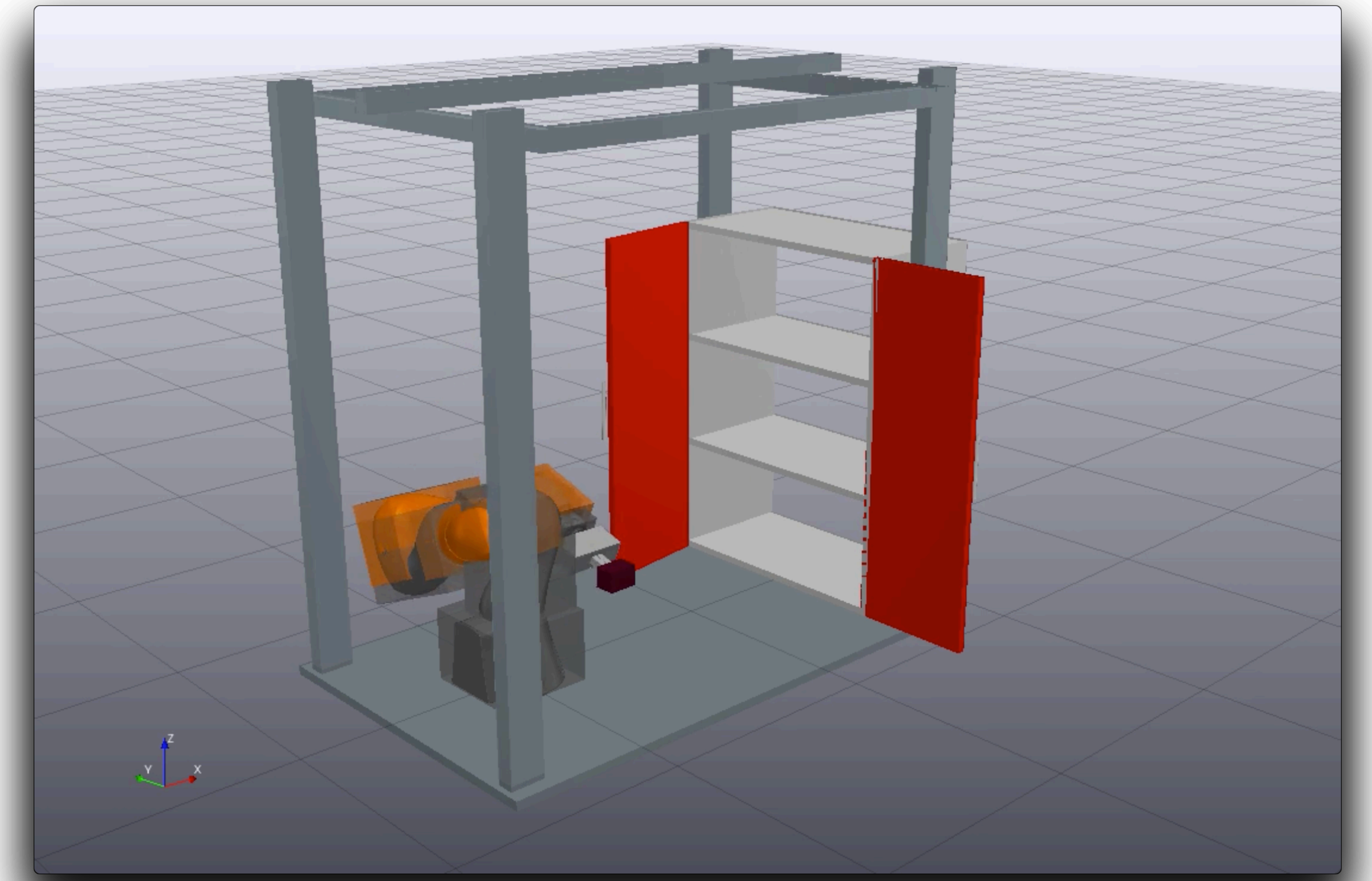


# PDDLStream + Drake Actions



- Problem formulation is **very similar** to the 2D pick-and-place example
  - Configurations & poses simply have **more DOFs**

```
(:action place
  :parameters (?r ?o ?p ?g ?q ?t)
  :precondition (and (Kin ?r ?o ?p ?g ?q ?t)
                     (AtGrasp ?r ?o ?g)
                     (AtConf ?r ?q)
                     (not (UnsafeTraj ?t)))
  :effect (and (AtPose ?o ?p)
               (HandEmpty ?r)
               (not (AtGrasp ?r ?o ?g))))

(:action pull
 … ; TODO: implementation
```



# PDDLStream + Drake Streams



- Stream implementations have similar declarations but more complex Python implementations
  - **Randomize initial guess** to sample many solutions

```
(:stream inverse-kinematics
  :inputs (?r ?o ?p ?g)
  :domain (and (Robot ?r) (Pose ?o ?p) (Grasp ?o ?g))
  :outputs (?q ?t)
  :certified (and (Conf ?r ?q) (Traj ?t) (Kin ?r ?o ?p ?g ?q ?t)))
```

```python
from pydrake.multibody.inverse_kinematics import InverseKinematics
def get_ik_gen_fn(diagram, context, plant, scene_graph):
    def get_fn(robot_name, obj_name, obj_pose, obj_grasp):
        robot_model = plant.GetModelInstanceByName(robot_name)
        obj_model = plant.GetModelInstanceByName(obj_name)
        while True:
            ik_scene = InverseKinematics(plant)
            ... # TODO: implementation
            yield (robot_conf, robot_traj)
    return get_fn
```

# Cost-Sensitive Planning



- Actions may have costs specified as **nonnegative functions**

```
(:action move
  :parameters (?q1 ?t ?q2)
  :precondition (and (Motion ?q1 ?t ?q2)
                     (AtConf ?q1))
  :effect (and (AtConf ?q2)
               (not (AtConf ?q1)))
               (increase (total-cost) (Length ?t)))
```

- Function specification similar to derived predicates

```
(:function (Length ?t)
   (Traj ?t))
```

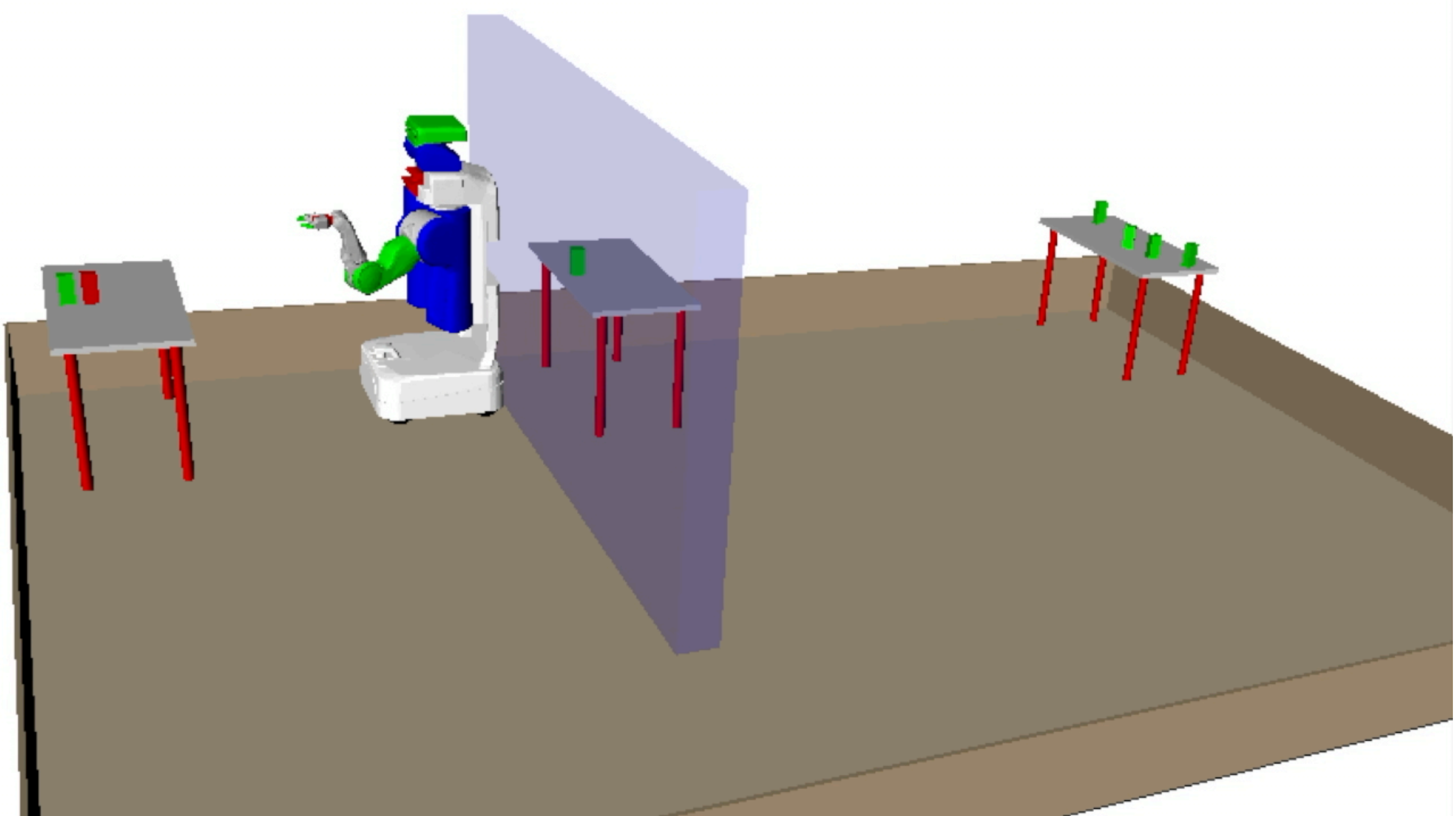
```
(:function (Distance ?q1 ?q2)
   (and (Conf ?q1) (Conf ?q2))
```

```
def Length(t):
    return sum(np.linalg.norm(q2 - q1)
               for q1, q2 in zip(t[:-1], t[1:]))
```

# Goal: Hold Any Green Block



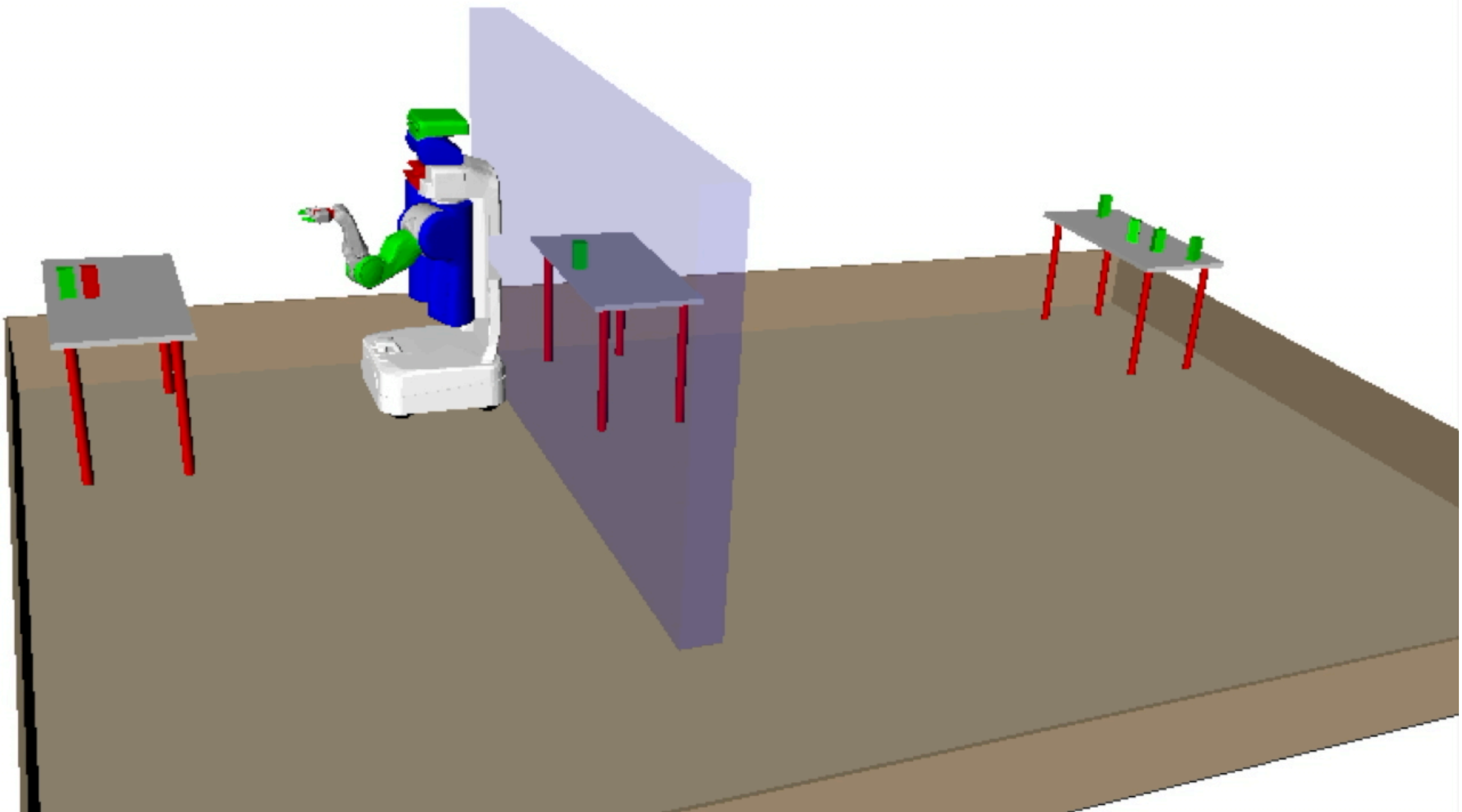
- **Lower bounds on costs** improve focused performance



# Goal: Hold Any Green Block



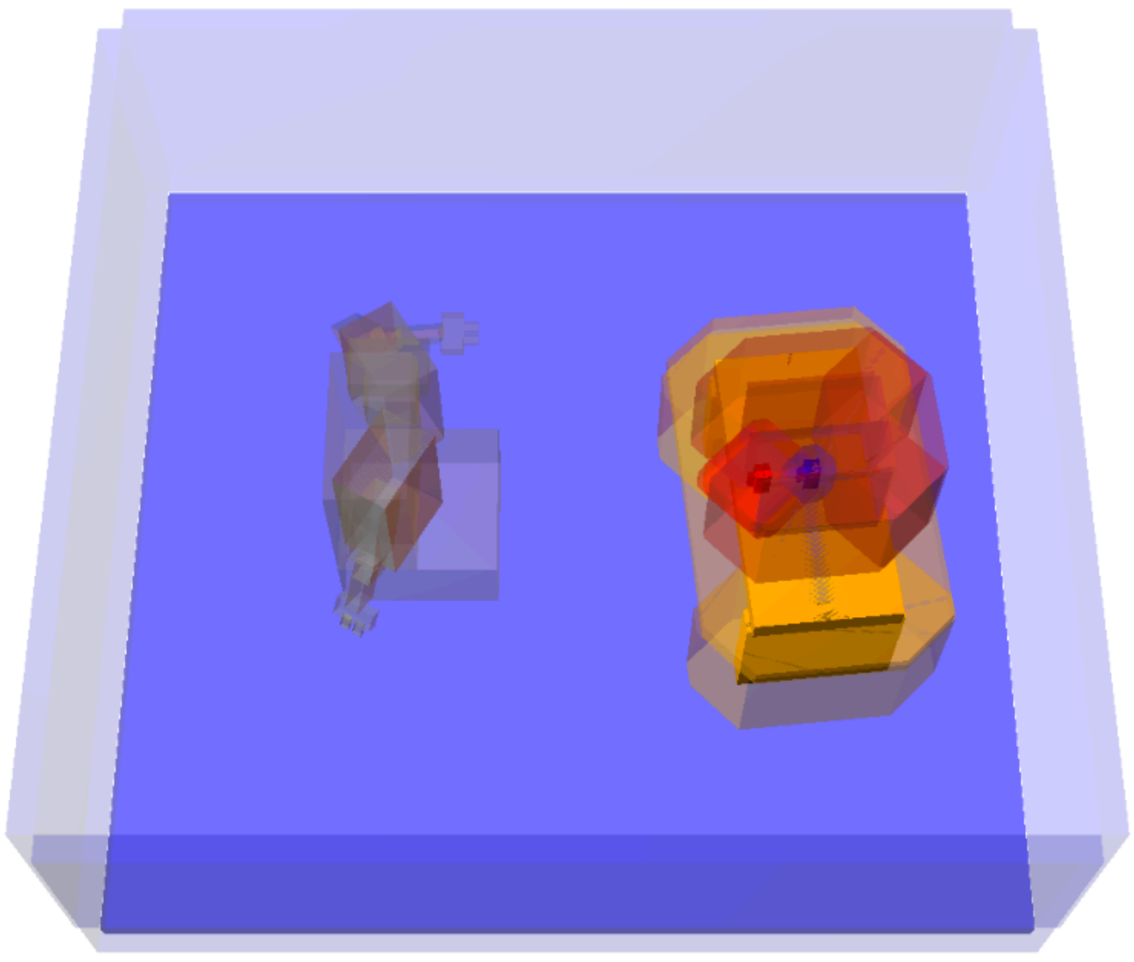
- **Lower bounds on costs** improve focused performance

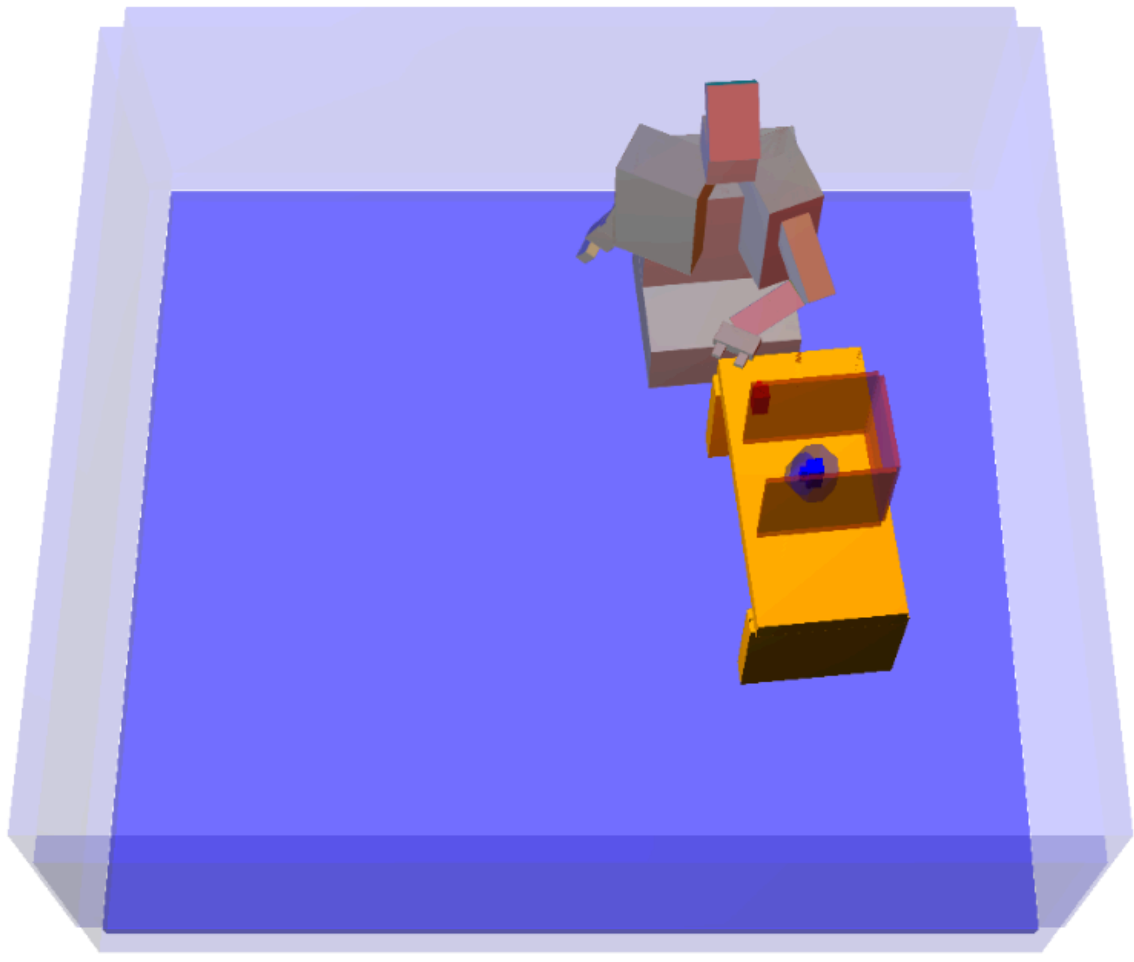


# Planning in the Real World



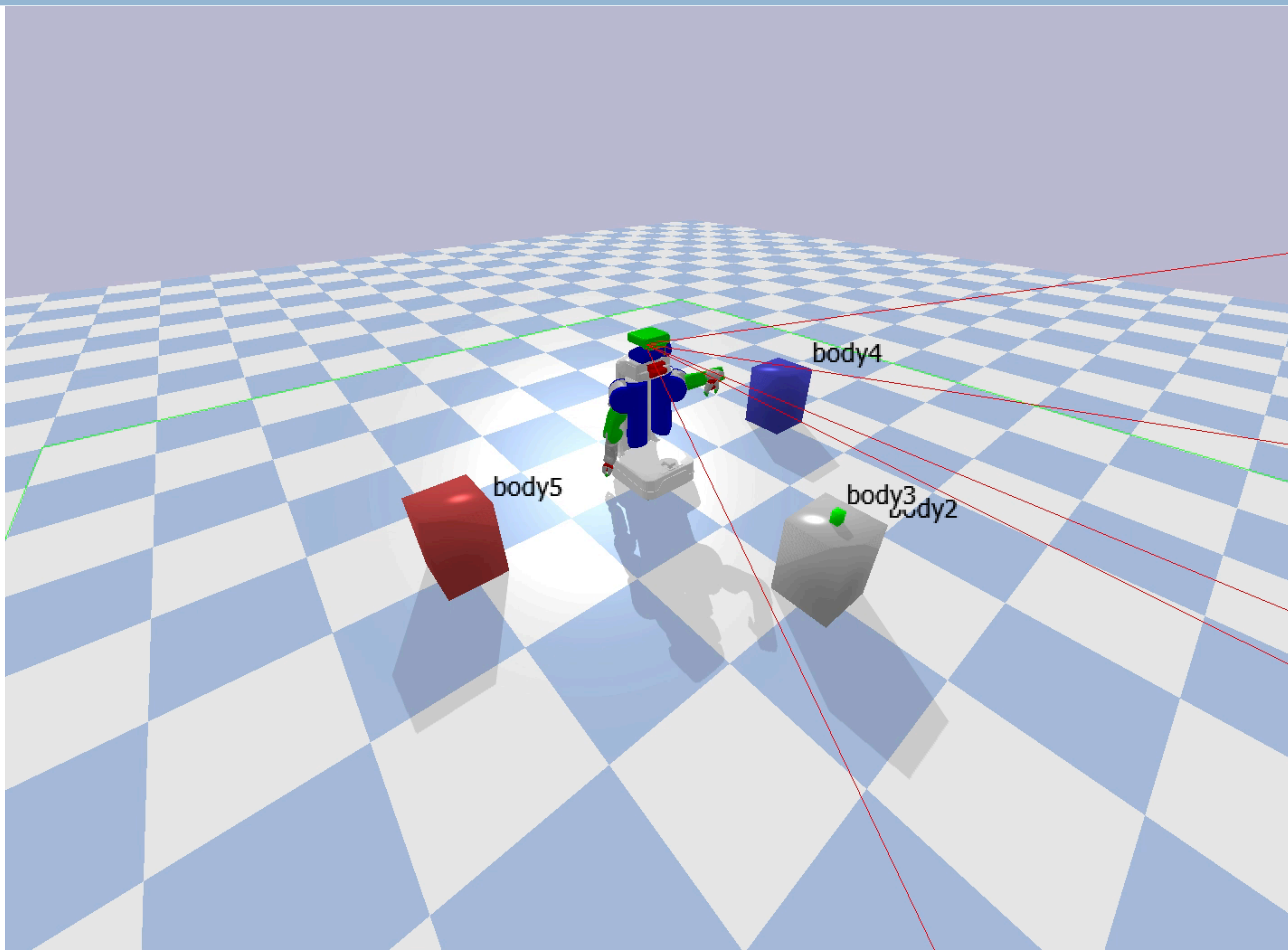
- **Nondeterministic outcomes -** stochastic effects
- **Partial observability**
  - The true state is unknown. Must perform inference.
- **Belief space planning (11/29/18)**
  - Update and control a probability distribution over states *[Platt, Kaelbling, Lozano-Perez, Tedrake]*





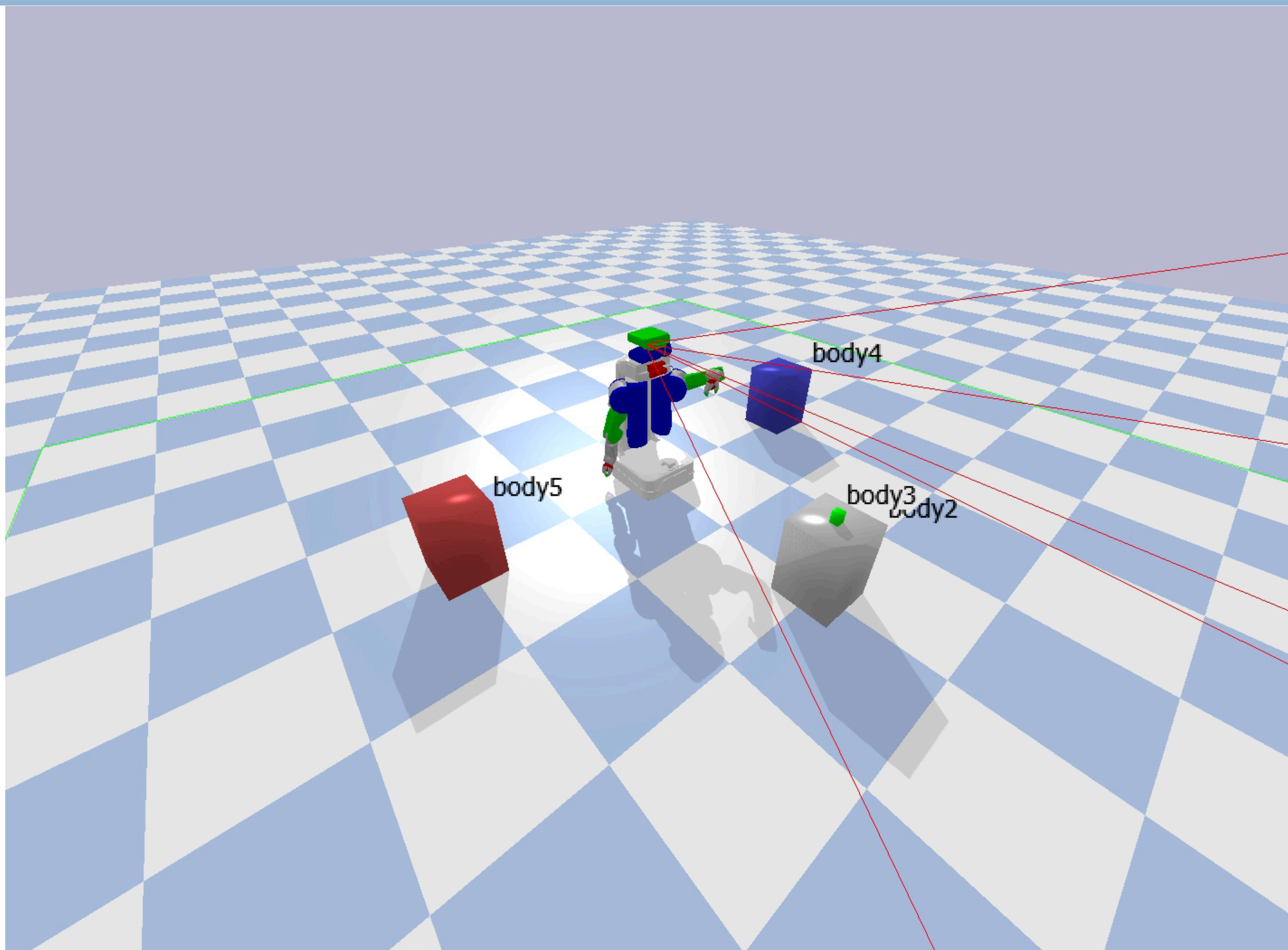
# Observing, Planning, & Executing





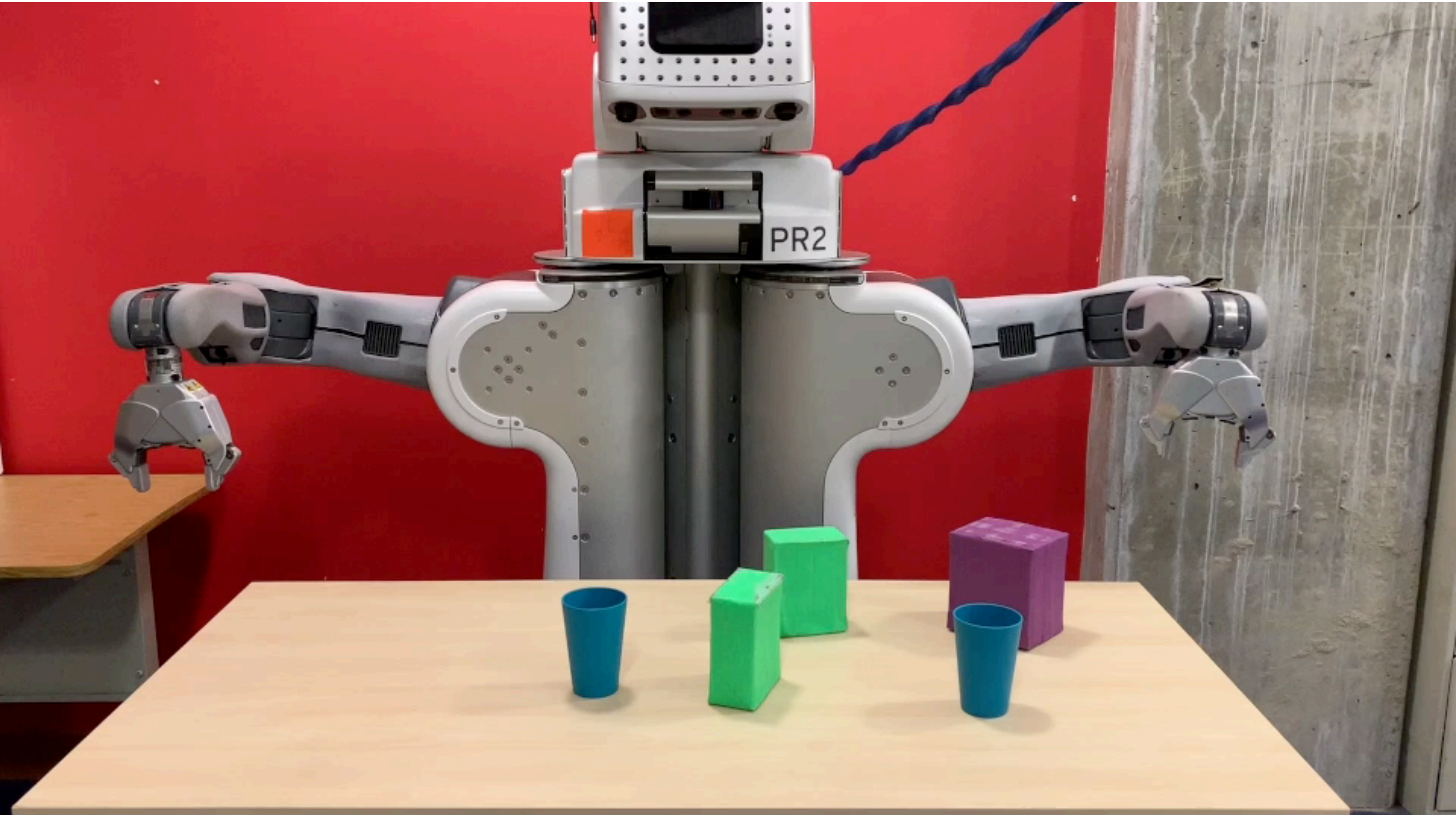
# Observing, Planning, & Executing





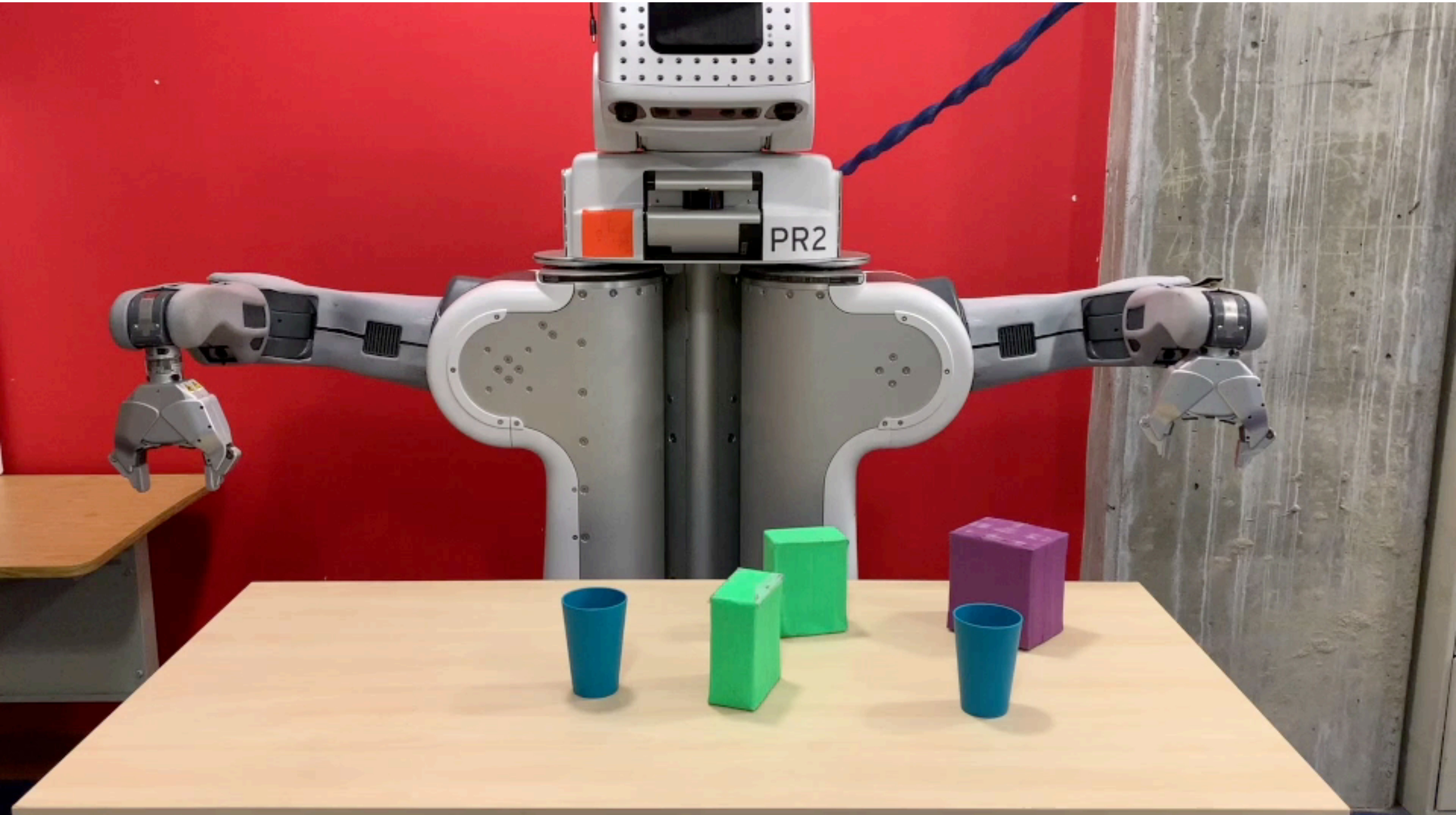
# PR2 Manipulation Outtakes





# PR2 Manipulation Outtakes





# PR2 Manipulation Outtakes