Lecture 27

CS 6380 Artificial Intelligence

Planning Beyond the Classical Setting

Department of Computer Science and Engineering

Indian Institute of Technology Madras



So Far in This Course

Search

- Uninformed search
- Informed search, A*

→

Classical Planning

- States as facts
- STRIPS actions

→

PDDL

- Domain and problem
- Planning. Domains

→

Heuristics

- Planning graph
- Max level, level sum

Today we drop the simplifying assumptions one at a time and see what new problems, representations and algorithms appear

This is also a preview of **CS6260 Automated Planning and Learning**, offered next semester

Classical Planning Makes Strong Assumptions

1

Fully observable

The agent knows the exact state

2

Deterministic

Each action has one outcome

3

Instant actions

No durations, no deadlines

4

No resources

Nothing is used up or shared

5

Flat actions

One level of primitive actions

6

Single agent

No one else changes the world

7

Known model

Someone wrote correct PDDL

8

Offline

Plan fully, then execute blindly

Real planners for spacecraft, factories and logistics have to give up most of these

Relaxing Each Assumption Opens a New Area

Assumption dropped	Area of planning	Representation (in CS6260)
Instantaneous actions, no resources	Temporal planning and scheduling	Deterministic planning, PDDL 2.1
Flat actions	Hierarchical (HTN) planning	Hierarchical planning, HDDL
Deterministic, fully observable	Conformant, contingent, FOND, MDPs	Planning under uncertainty
Single agent	Multiagent planning	MA-PDDL
Known model	Learning for planning	Learning models, heuristics, methods
Purely symbolic world	Robotics, RL and LLM integration	Practical applications

Time, Schedules, and Resources

When actions take time and use things up

What Classical Plans Cannot Say

- Classical plans say **what** to do and **in what order**
- They cannot say **how long** an action takes or **when** it occurs
  - An airline needs departure and arrival times, not just which plane flies which route
- The real world adds **resource constraints**
  - Limited staff, and staff on one flight cannot be on another
- Common approach is **plan first, schedule later**

Planning phase

Select actions and the ordering constraints needed to reach the goal



Scheduling phase

Add start times so that durations, resources and deadlines are respected

Job Shop Scheduling

- A set of **jobs**, each a sequence of **actions** with ordering constraints
- Each action has a **duration** and resource needs
  - **Consumable** resources are used up, like lug nuts
  - **Reusable** resources are borrowed and returned, like an engine hoist
- Goal is to minimise the **makespan**, the total duration of the plan
- **Aggregation** counts identical objects, Inspectors(2), instead of naming each one
  - Avoids trying all 10! assignments of inspectors to actions

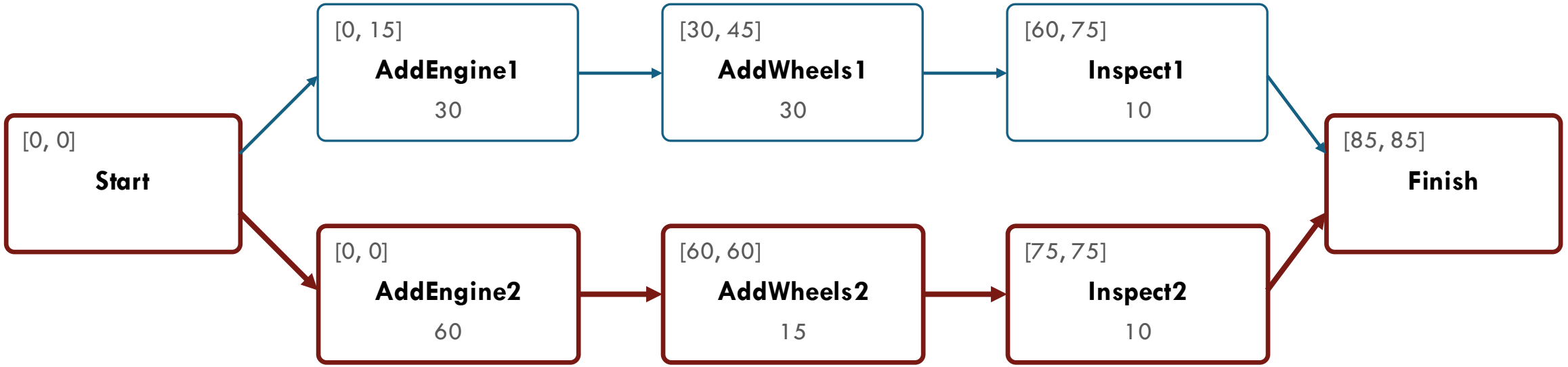
```
Jobs({AddEngine1 ≺ AddWheels1 ≺ Inspect1},
     {AddEngine2 ≺ AddWheels2 ≺ Inspect2})

Resources(EngineHoists(1), WheelStations(1),
          Inspectors(2), LugNuts(500))

Action(AddEngine1, DURATION 30, USE EngineHoists(1))
Action(AddEngine2, DURATION 60, USE EngineHoists(1))
Action(AddWheels1, DURATION 30,
       CONSUME LugNuts(20), USE WheelStations(1))
Action(AddWheels2, DURATION 15,
       CONSUME LugNuts(20), USE WheelStations(1))
Action(Inspect_i, DURATION 10, USE Inspectors(1))
```

Assembling two cars. A ≺ B means A must come before B

Critical Path Method



- Each node shows **[ES, LS]**, the earliest and latest possible start time
- The **critical path** is the longest path, here 85 minutes. Delaying any action on it delays the whole plan
- **Slack** of an action is LS minus ES. Actions on the critical path have zero slack
- ES and LS for all actions together form a **schedule**

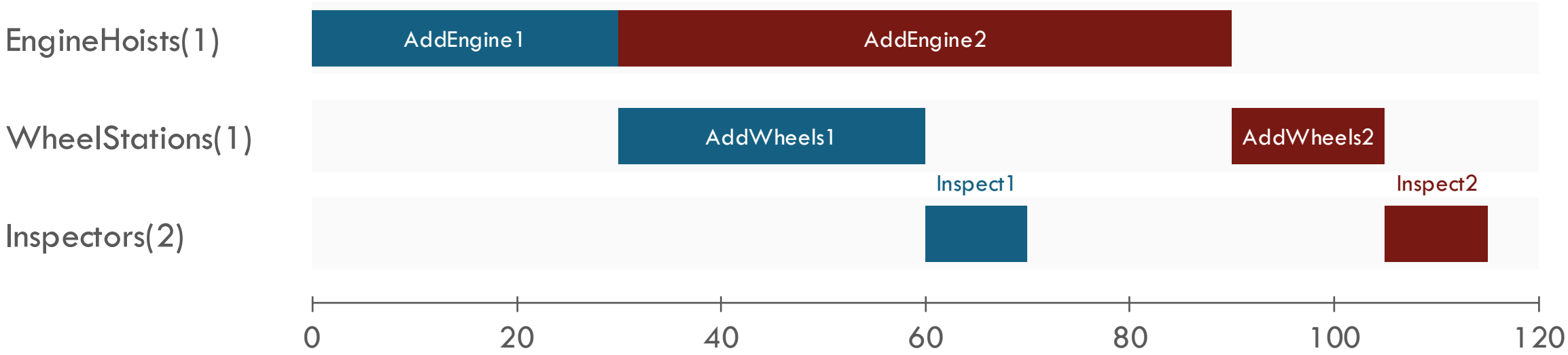
Computing the Schedule

$$ES(Start) = 0$$
$$ES(B) = \max\ \{\ ES(A) + Duration(A)\ \}\ \text{over}\ A \prec B$$
$$LS(Finish) = ES(Finish)$$
$$LS(A) = \min\ \{\ LS(B) - Duration(A)\ \}\ \text{over}\ B \succ A$$

- Forward pass for ES, backward pass for LS
- Runs in **O(Nb)** for N actions and branching factor b
- Easy because all constraints are a **conjunction of linear inequalities**

Action	ES	LS	Slack
AddEngine1	0	15	15
AddWheels1	30	45	15
Inspect1	60	75	15
AddEngine2	0	0	0
AddWheels2	60	60	0
Inspect2	75	75	0
Finish	85	85	0

Adding Resource Constraints



- AddEngine1 and AddEngine2 share one hoist, so they **cannot overlap**. That is a disjunction of two inequalities, and the problem becomes **NP-hard**
- Best schedule takes **115** minutes instead of 85
- A 1963 challenge with 10 machines and 10 jobs of 100 actions stayed unsolved for **23 years**
- The **minimum slack** heuristic (schedule the action with least slack first) is popular, but gives 130 here

Time in PDDL, Durative Actions

```
(:durative-action add-wheels
  :parameters (?c - car ?s - station)
  :duration (= ?duration 30)
  :condition (and
    (at start (engine-added ?c))
    (at start (free ?s))
    (over all (at ?c ?s)))
  :effect (and
    (at start (not (free ?s)))
    (at end (free ?s))
    (at end (wheels-added ?c))
    (at start (decrease (lug-nuts) 20))))
```

- **PDDL 2.1** adds durations and numeric fluents
- Conditions and effects are tagged in time
  - **at start**, at the moment the action begins
  - **over all**, throughout its execution
  - **at end**, when it finishes
- Numeric fluents model consumable resources
- Temporal planners interleave **choosing actions** and **scheduling them**

Hierarchical Planning

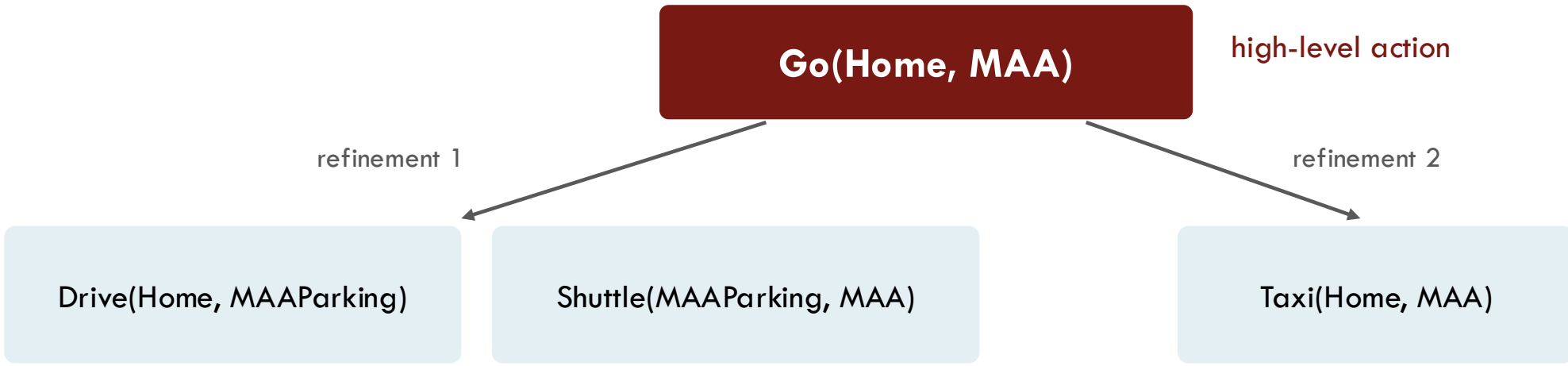
Planning at many levels of abstraction

Why Hierarchy

10^3	$10 / s$	10^9	10^{13}
muscles to activate	activations per muscle	seconds awake in a life	primitive actions in a lifetime

- Even a two week vacation in Lakshadweep would need about 10^{10} motor actions
- Humans plan abstractly, like **go to Chennai airport, fly to Agatti via Kochi, relax by the lagoon, fly back, go home**
- Each abstract step is refined later, some only during execution, like finding a parking spot
- **Hierarchical decomposition** reduces each task to a small number of activities at the next lower level

High-Level Actions and Refinements



- **Primitive actions** have precondition and effect schemas, as in PDDL
- A **high-level action (HLA)** has one or more **refinements** into sequences of HLAs or primitives
- A refinement with only primitives is an **implementation**

```
Refinement(Navigate([a,b], [x,y]),
  PRECOND Connected([a,b], [a-1,b])
  STEPS [Left, Navigate([a-1,b], [x,y])])

Refinement(Navigate([a,b], [x,y]),
  PRECOND a = x ∧ b = y,  STEPS [ ])
```

Refinements can be recursive

When Does a High-Level Plan Work

A high-level plan achieves the goal from a state if **at least one** of its implementations achieves the goal from that state

HTN planning

The **agent** chooses which implementation to run

One working implementation is enough

Nondeterministic planning

Nature chooses the outcome

The plan must work for all outcomes

- If an HLA has exactly one implementation, we can compute its preconditions and effects and treat it like a primitive action
- The right set of HLAs can make blind search **linear** instead of exponential in solution depth

Searching for Primitive Solutions

```
function HIERARCHICAL-SEARCH(problem, hierarchy)
  frontier ← FIFO queue with [Act]
  loop
    if EMPTY(frontier) then return failure
    plan ← POP(frontier)
    hla ← first HLA in plan, or null
    prefix, suffix ← actions before, after hla
    outcome ← RESULT(initial state, prefix)
    if hla is null then
      if outcome satisfies goal then return plan
    else for each seq in REFINEMENTS(hla, outcome)
      frontier ← INSERT(prefix + seq + suffix)
```

- Start from a single top level action **Act**
- Repeatedly replace an HLA with one of its refinements
- Stop when the plan is primitive and reaches the goal
- Classical planning is a special case
  - Act refines to [a_i, Act] for every action a_i, or to nothing when the goal holds
- Domain knowledge lives in the **HLA library**

Why Hierarchy Pays Off

Flat forward search $O(b^d)$

Regular hierarchy $r^{(d-1)/(k-1)}$

- r refinements per HLA, each with k steps
- Small r and large k gives roughly the **k-th root** of the flat cost
- Long reusable action sequences are precious

O-PLAN at Hitachi

350
products

35
assembly machines

2000+
operations

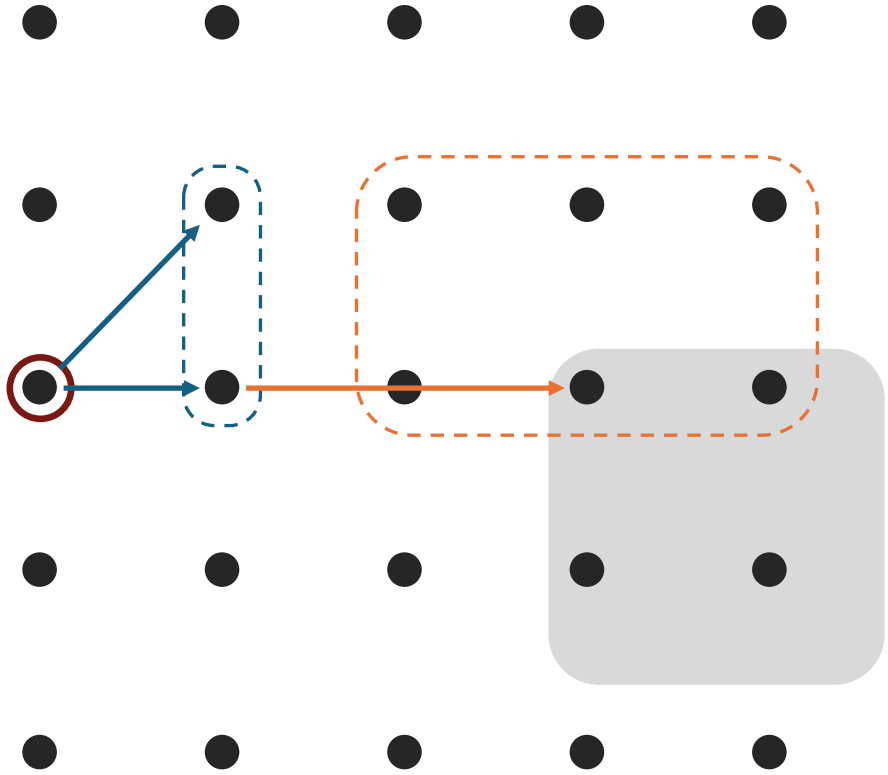
30 days
three shifts per day

HTN planning plus scheduling produces plans with tens of millions of steps

Angelic Semantics and Reachable Sets

- Goal is to prove a high-level plan works **without refining it**
- Needs the **downward refinement property**, every plan that claims to work has a working implementation
- **Reachable set** REACH(s, h) is the set of states reachable by any implementation of h
- A high-level plan works if its reachable set **intersects** the goal

$$\text{REACH}(s, [h_1, h_2]) = \bigcup \text{REACH}(s', h_2),\ s' \in \text{REACH}(s, h_1)$$



s in red, REACH(s, h_1) in blue,
REACH after h_2 in orange, goal shaded

Optimistic and Pessimistic Descriptions

$$\text{REACH}^{-}(s, h) \subseteq \text{REACH}(s, h) \subseteq \text{REACH}^{+}(s, h)$$

✗

Plan fails

Optimistic set REACH^{+} misses the goal

✓

Plan works

Pessimistic set REACH^{-} intersects the goal

?

Refine further

Only the optimistic set reaches the goal

- **Angelic search** commits to a working abstract plan, then decomposes it into subproblems
- Cleaning a large vacuum world, breadth first grows as 5^d while angelic search is roughly **linear** in the number of squares

Hierarchical Planning in CS6260

```
(:method m-go-by-taxi
  :parameters (?from ?to - location)
  :task (go ?from ?to)
  :ordered-subtasks (and
    (call-taxi ?from)
    (ride-taxi ?from ?to)))

(:method m-go-by-car
  :parameters (?from ?to - location ?p -
parking)
  :task (go ?from ?to)
  :ordered-subtasks (and
    (drive ?from ?p) (shuttle ?p ?to)))
```

Two methods for the same task in HDDL

- **Tasks** and **methods** instead of HLAs and refinements
- **HDDL**, the hierarchical extension of PDDL
- Total order and partial order HTN planning
- HTN planners such as SHOP2, PANDA and Lilotane
- Acting with HTNs, refining during execution
- Learning HTN methods from experience

Planning Under Uncertainty

Partial observability, nondeterminism, and wrong models

Ways to Handle Uncertainty

1

Sensorless

Conformant planning

No observations. Find one plan that works in every possible state

2

Contingent

Conditional plans

Branch on what is perceived during execution

3

Online

Replanning

Monitor execution and repair the plan when things go wrong

4

Probabilistic

MDPs and PPDDL

Outcomes have probabilities. Find a policy that maximises expected reward

Planners work with **belief states,** the sets of physical states the agent might be in

Running Example, Painting a Chair and Table

```
Init(Object(Table) ∧ Object(Chair) ∧ Can(C1) ∧
Can(C2)
     ∧ InView(Table))
Goal(Color(Chair, c) ∧ Color(Table, c))

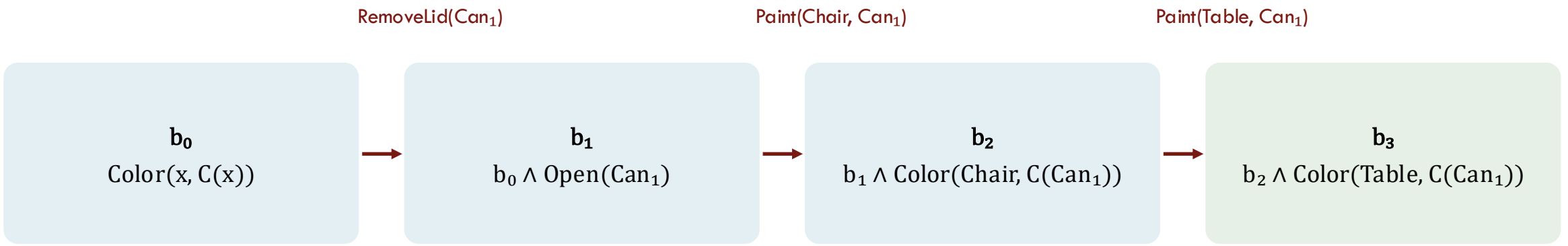
Action(RemoveLid(can), PRECOND Can(can),
       EFFECT Open(can))
Action(Paint(x, can),
       PRECOND Object(x) ∧ Can(can) ∧ Color(can, c)
∧ Open(can),
       EFFECT Color(x, c))
Action(LookAt(x), PRECOND InView(y) ∧ (x ≠ y),
       EFFECT InView(x) ∧ ¬InView(y))

Percept(Color(x, c), PRECOND Object(x) ∧ InView(x))
Percept(Color(can, c),
        PRECOND Can(can) ∧ InView(can) ∧ Open(can))
```

- Goal is for chair and table to have the **same color**
- Colors of paint and furniture are unknown
- **Percept schemas** say when the agent observes a fluent
- Fully observable means percepts with no preconditions
- Sensorless means no percept schemas at all

Sensorless Planning

- Switch from the closed world assumption to an **open world assumption.** Fluents not mentioned are unknown
- Belief state update for a deterministic action is $b' = (b - \text{DEL}(a)) \cup \text{ADD}(a)$



- b_3 satisfies the goal with c bound to $C(Can_1)$. The agent **coerces** the world into a goal state without knowing the color
- Belief states that are conjunctions of literals (**1-CNF**) stay 1-CNF under PDDL updates, so n fluents need only O(n) space, not 2^n

Keeping Belief States Manageable

- **Conditional effects** make an effect depend on the state, and break 1-CNF
- Suck in the vacuum world has effect **when** AtL then CleanL, **when** AtR then CleanR
- From b = True we get (AtL ∧ CleanL) ∨ (AtR ∧ CleanR)

Choose actions that keep beliefs simple

Action	Belief state
start	True
Right	AtR
Suck	AtR ∧ CleanR
Left	AtL ∧ CleanR
Suck	AtL ∧ CleanR ∧ CleanL

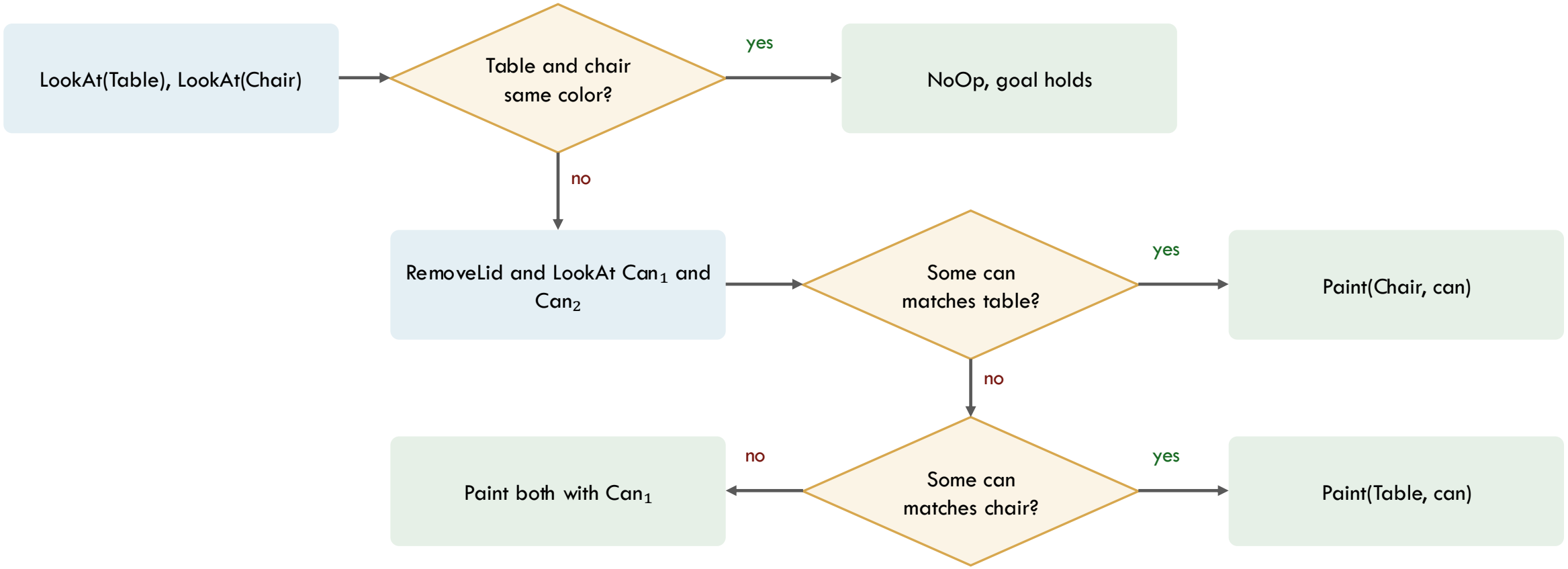
Heuristics for belief states

$$\text{If } b_1 \subseteq b_2 \text{ then } h^*(b_1) \leq h^*(b_2)$$

$$H(b) = \max\{ h(s_1), \ldots, h(s_N) \}$$

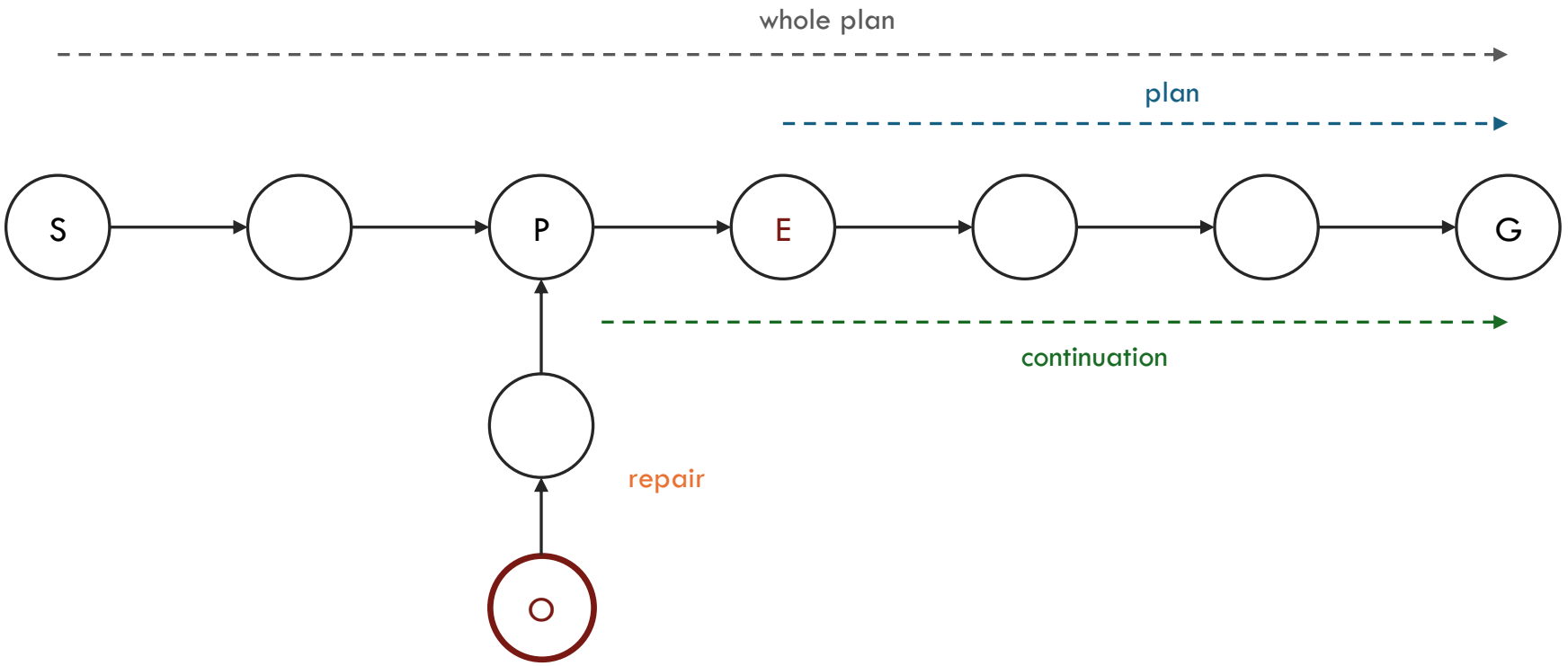
- Solving a subset of a belief state is never harder
- So any admissible h on sampled states gives an admissible H
- A planning graph can start from a 1-CNF belief state
- Ignore delete lists works well in practice

Contingent Planning



- Belief state is updated in two stages, first for the action and then for the percept
- Plans are found with **AND-OR search** over belief states. Nondeterministic effects are just disjunctions in the effect

Online Replanning and Execution Monitoring



Expected to be in **E**, observed **O**. Repair back to **P** on the plan, then continue to **G**

Action monitoring

Before each action, check its preconditions still hold

Plan monitoring

Check the rest of the plan will still succeed. Catches doomed plans early and allows serendipity

Goal monitoring

Check whether there is a better set of goals to pursue

Why Plans Fail, Wrong Models

Missing precondition

Removing a lid may need a screwdriver

Missing effect

Painting may also put paint on the floor

Missing state variable

The model has no notion of how much paint is left

Exogenous events

Someone knocks over the paint can

Retrying a repair that depends on an unknown precondition can loop forever, like painting from an empty can

Every prediction failure is an opportunity to learn a better model

Planning Under Uncertainty

MDPs

- States, actions, transition probabilities, rewards
- Value iteration and policy iteration
- Link to reinforcement learning

FOND planning

- Fully observable, nondeterministic effects
- Weak, strong and strong cyclic solutions
- Solutions are policies, not sequences

PPDDL

- PDDL with probabilistic effects
- Learning PPDDL models from data
- Noisy deictic rules (NDRs)

$$V(s) = \max_a \Sigma_{s'} T(s, a, s') [R(s, a, s') + \gamma V(s')]$$

Bellman equation for the optimal value function

```
(:action pick-up
  :parameters (?b - block)
  :precondition (and (clear ?b) (handempty))
  :effect (probabilistic
     0.8 (holding ?b)
     0.2 (on-table ?b)))
```

A slippery gripper in PPDDL