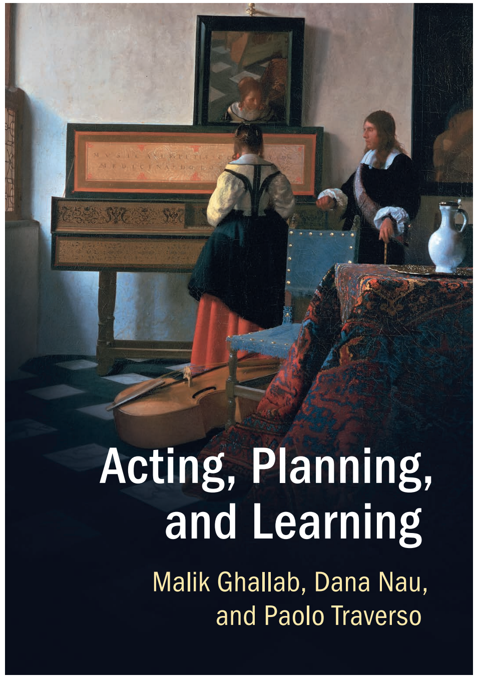
Last update: 10:55 AM, March 8, 2025



Chapter 2

Deterministic Representation and Acting

Dana S. Nau
University of Maryland

with contributions from
Mark “mak” Roberts

Lecture slides for *Acting, Planning, and Learning*. Creative Commons CC BY-SA 4.0

Motivation

- How to model a complex environment?
  - Generally need simplifying assumptions
- *Classical planning*
    - Finite, static world, just one actor
    - No concurrent actions, no explicit time
    - Determinism, no uncertainty, no exogeneous events
    - Full observability
    - Unit-cost actions
  - Sequence of states and actions $\langle s_0, a_1, s_1, a_2, s_2, \ldots\rangle$
- Avoids many complications
- Most real-world environments don’t satisfy the assumptions

  ⇒ Errors in prediction
- OK if they’re infrequent and don’t have severe consequences

Outline

2.2. State-Transition Systems

2.3. State-Variable Representation

2.6. Acting

2.4. Classical Representation

2.5. Computational Complexity

Chapter 2 of Haslum *et al*. (2019)*

- Classical fragment of PDDL
- Planning domains and problems
- untyped, typed

---

* Haslum, Lipovetzky, Magazzini, & Muise. *An Introduction to the Planning Domain Definition Language*. Morgan Claypool, 2019.

Section 2.1. State-Transition Systems

State-transition system or *classical planning domain*:

- $\Sigma = (S, A, \gamma, \text{cost})$ or (S, A, γ)
  - S - finite set of *states*
  - A - finite set of *actions*
  - $\gamma: S \times A \rightarrow S$

    prediction (or *state-transition*) function
    - *partial* function: $\gamma(s,a)$ is not necessarily defined for every (s,a)
      - a is *applicable* in s iff $\gamma(s,a)$ is defined
      - $\text{Domain}(a) = \{s \in S \mid a \text{ is applicable in } s\}$
      - $\text{Range}(a) = \{\gamma(s,a) \mid s \in \text{Domain}(a)\}$
  - $\text{cost}: S \times A \rightarrow \mathbb{R}^+$ or $\text{cost}: A \rightarrow \mathbb{R}^+$
    - optional; default is $\text{cost}(a) \equiv 1$
    - money, time, something else

- *plan*:
  - a sequence of actions $\pi = \langle a_1, \ldots, a_n \rangle$
- π is *applicable* in s_0 if the actions are applicable in the order given

$$
\begin{aligned}
&\gamma(s_0, a_1) = s_1 \\
&\gamma(s_1, a_2) = s_2 \\
&\ldots \\
&\gamma(s_{n-1}, a_n) = s_n
\end{aligned}
$$

  - In this case define $\gamma(s_0, \pi) = s_n$

- *Classical planning problem*:
  - $P = (\Sigma, s_0, S_g)$
  - planning domain, initial state, set of goal states
- *Solution* for P:
  - a plan π such that that $\gamma(s_0, \pi) \in S_g$

Planning Problems

- $\pi = \langle a_1, \ldots, a_n \rangle$ is *applicable* in s_0 if the actions are applicable in the order given

$$\gamma(s_0, a_1) = s_1$$
$$\gamma(s_1, a_2) = s_2$$
$$\ldots$$
$$\gamma(s_{n-1}, a_n) = s_n$$

  - In this case we define
    - $\gamma(s_0, \pi) = s_n$
    - $\hat{\gamma}(s_0, \pi) = \langle s_0, \ldots, s_n \rangle$

- *Classical planning problem*:
  - $P = (\Sigma, s_0, S_g)$
  - planning domain, initial state, set of goal states

- *Solution* for P: a plan π such that that $\gamma(s_0, \pi) \in S_g$
  - *Minimal solution:* no subsequence is also a solution
  - *Shortest solution:* no solution has fewer actions
  - *Optimal solution:* no solution has lower cost

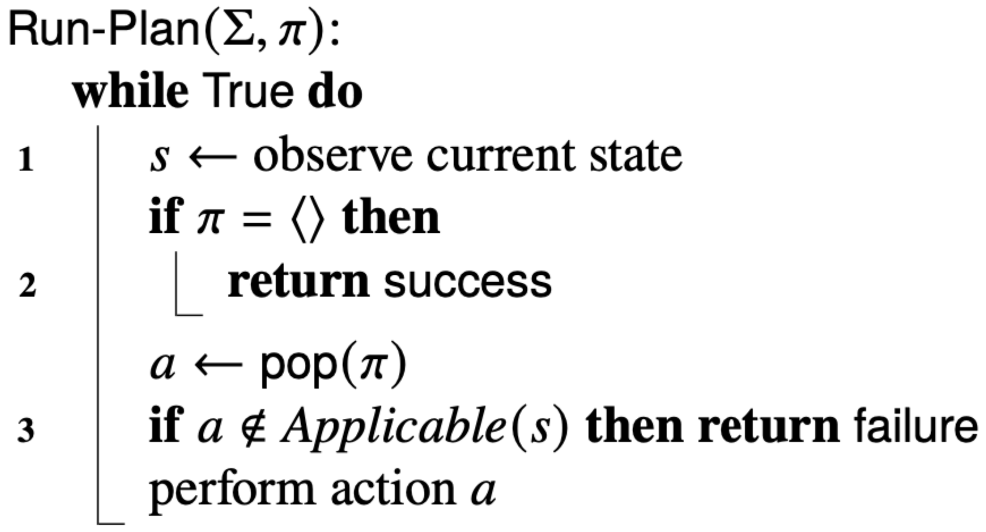
- **Example:** Suppose P has three solutions
  - $\pi_1 = \langle a_1 \rangle$
  - $\pi_2 = \langle a_2, a_3, a_4, a_5 \rangle$
  - $\pi_3 = \langle a_2, a_3, a_1 \rangle$
  - Then π_1 is both shortest and optimal
- **Poll:** Which solutions are minimal?

A. π_1 B. π_2 C. π_3

Lecture slides for *Acting, Planning, and Learning*. Creative Commons CC BY-SA 4.0

Acting with a Plan

- A simple procedure for running a plan



```
Run-Plan($\Sigma, \pi$):
    while True do
1       $s \leftarrow$ observe current state
        if $\pi = \langle\rangle$ then
2           return success
        $a \leftarrow$ pop($\pi$)
3       if $a \notin Applicable(s)$ then return failure
        perform action $a$
```

- To test whether π has achieved a desired goal S_g
  - add S_g as a third argument
  - before line 2, insert this:

    if $s \notin S_g$ **then return** failure

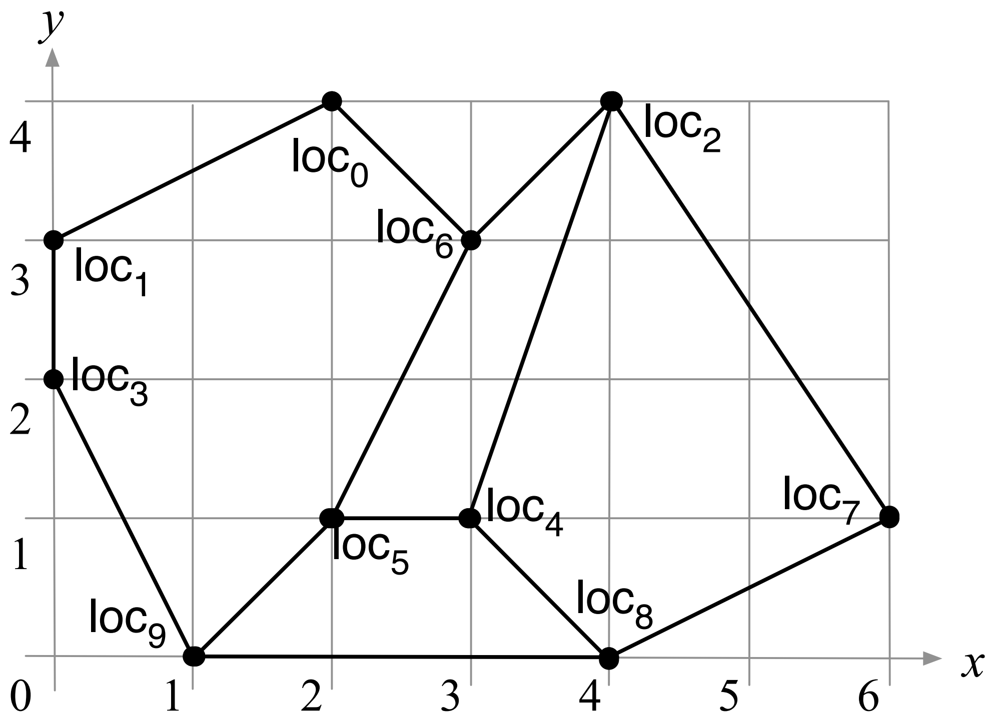
- Ideally, Run-Plan(Σ, $\langle a_1, \ldots, a_n \rangle$) will take Σ through through the sequence of states
  - $\hat{\gamma}(s_0, \pi) = \langle s_1, \ldots, s_n \rangle$

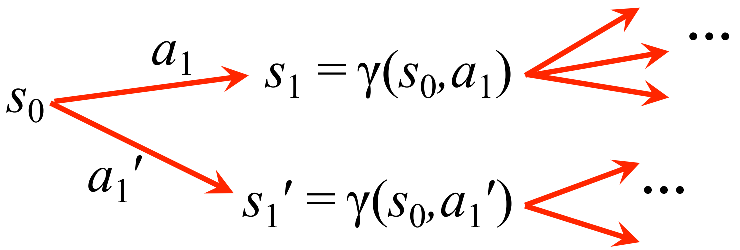
  then return success

- But recall that Σ is unlikely to be a perfect model of the actor's environment
  - Later we'll discuss some things that can go wrong

Section 2.2. Representation

- We write Run-Plan(Σ, π)
  - But what Run-Plan really needs is data structures that represent Σ and π
- If S and A are small enough
  - Give each state and action a name
  - For each s and a, store $\gamma(s,a)$ in a lookup table
- In larger domains, don't represent all states explicitly
  - Language for describing properties of states
  - Language for describing how each action changes those properties
  - Start with initial state, use actions to produce other states





Lecture slides for *Acting, Planning, and Learning*. Creative Commons CC BY-SA 4.0

Kinds of Representations

- *Domain-specific* representation:
  - tailor-made for a specific environment
- State: arbitrary data structure
- Action: (head, preconditions, effects, cost)
  - *head*: name and parameter list
    - Get actions by instantiating the parameters
  - *preconditions*:
    - Computational tests to predict whether an action can be performed
    - Should be necessary/sufficient for the action to run without error
  - *effects*:
    - Procedures that modify the current state
  - *cost*: procedure that returns a number
    - Can be omitted, default is cost ≡ 1

- Advantage: can use whatever works best for that particular domain
- Disadvantage: for each new domain, need new representation, new algorithms

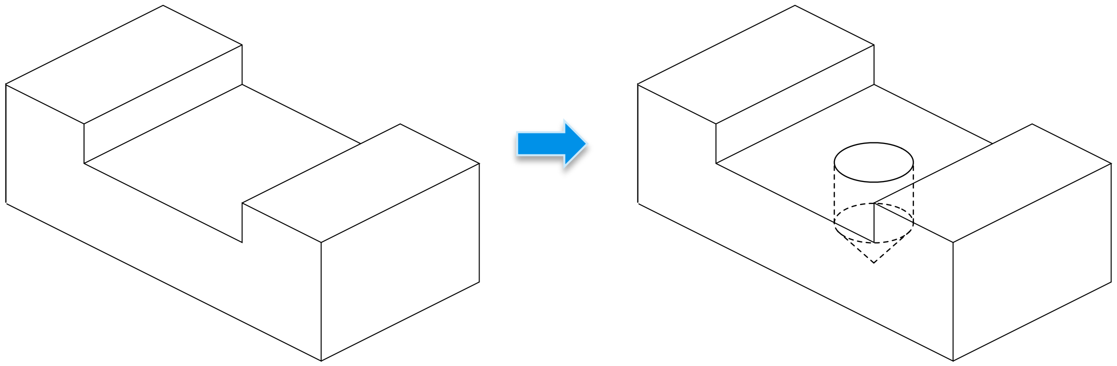
- Alternative: *domain-independent* representation
  - A “standard format” that can be used for many different planning domains
  - Limited representational capability, but easy to compute
  - Domain-independent algorithms that work for anything in this format

  - We’ll use a *state-variable* representation …

Creative Commons CC BY-SA 4.0

Example

- Drilling holes in a metal workpiece
  - A state
    - geometric model of the workpiece
      - *annotated* with dimensions, tolerances, etc.
    - capabilities and status of drilling machine and drill bit
  - Several actions
    - clamp the workpiece onto the drilling machine
    - load a drill bit into the machine
    - drill a hole



- Name: drill-hole
- Arguments:
  - ID codes for the machine and drill bit
  - annotated geometric model of the workpiece
  - description of the hole to be drilled
- Preconditions
  - *Capabilities*: can the machine and drill bit produce the desired hole?
  - *Current state*: Is the drill bit installed? Is the workpiece clamped onto the table? Etc.
- Effects
  - annotated geometric model of modified workpiece
- Cost
  - estimate of time or monetary cost

Lecture slides for *Acting, Planning, and Learning*. Creative Commons CC BY-SA 4.0

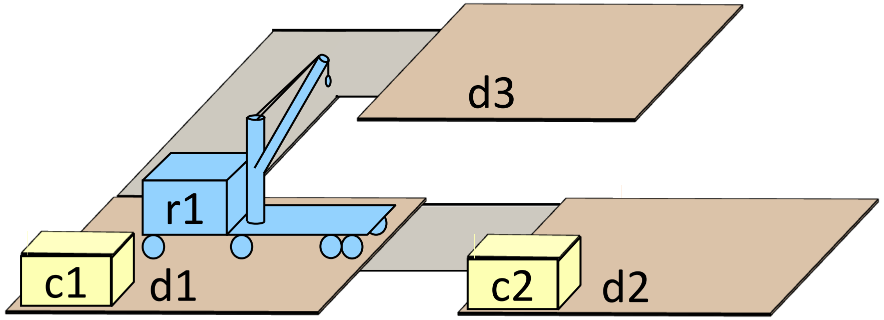
Discussion

- Advantage of domain-specific representation:
  - use whatever works best for that particular domain
- Disadvantage:
  - for each new domain, need new representation and deliberation algorithms
- Alternative: *domain-independent* representation
  - Try to create a “standard format” that can be used for many different planning domains
  - Deliberation algorithms that work for anything in this format

- *State-variable* representation
  - Simple formats for describing states and actions
  - Limited representational capability
    - But easy to compute, easy to reason about
  - Domain-independent search algorithms and heuristic functions that can be used in all state-variable planning problems

State-Variable Representation

- *Objects* = {names of objects in the environment}
- Organized into an *typed ontology*
  - sets of object types



- *Objects* = *Robots* ∪ *Containers* ∪ *Locs* ∪ {nil}
  - *Robots* = {r1}
  - *Containers* = {c1, c2}
  - *Locs* = {d1, d2, d3}

- *Objects* only needs to include objects that matter at the current level of abstraction
- Can omit lots of details
  - physical characteristics of robots, containers, loading docks, roads, …

Lecture slides for *Acting, Planning, and Learning*. Creative Commons CC BY-SA 4.0

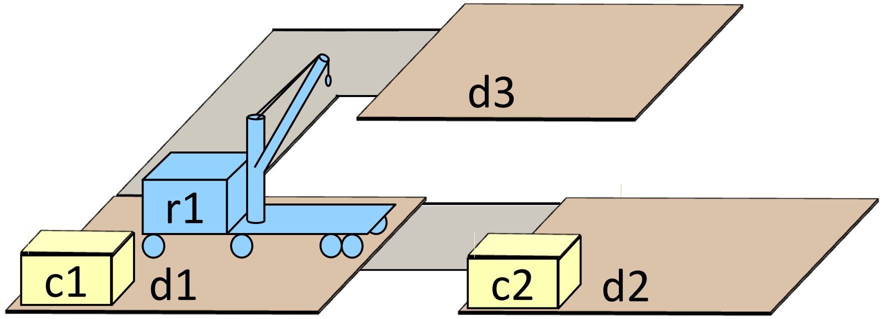
Rigid Properties

- Objects have two kinds of properties
  - *rigid* and *varying*
- *Rigid*: stays the same in every state
  - Can be described as a mathematical relation

    adjacent = {(d1,d2), (d2,d1), (d1,d3), (d3,d1)}
  - Or equivalently, a set of *ground atoms*

    adjacent(d1,d2), adjacent(d2,d1),

    adjacent(d1,d3), adjacent(d3,d1)
  - I'll use the two notations interchangeably



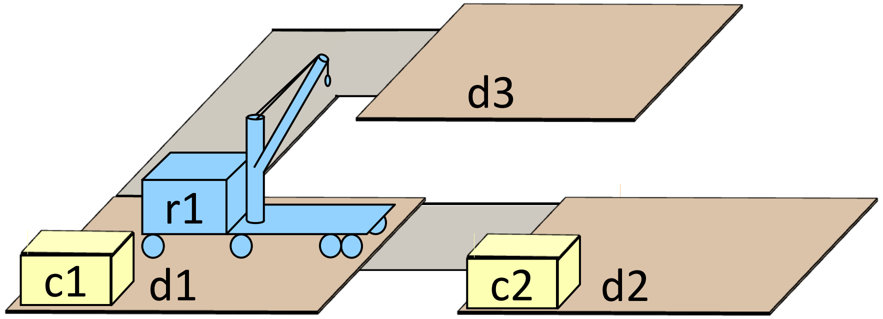
Terminology from first-order logic:

- *atom* ≡ *atomic formula* ≡ *positive literal* ≡ predicate symbol with list of arguments
  - *e.g.*, adjacent(x,d2), where x is unbound
- *negative literal* ≡ *negated atom* ≡ atom with negation sign in front of it
  - *e.g.*, ¬ adjacent(x,d2)
- an atom that contains no variable symbols is *ground* (or *fully instantiated*)
  - *e.g.*, adjacent(d1,d2)
- an atom that contains no constant symbols is *lifted*
  - *e.g.*, adjacent(x,y)
- an atom that contains both is *partially instantiated*
  - *e.g.*, adjacent(x,d2)
- *ground instance* of any expression: replace every variable with a value in its range
  - *e.g.*, adjacent(d1,d2) is a ground instance of both adjacent(x,d2) and adjacent(x,y)

Varying Properties

- *Varying* property (or *fluent*):
    - a property that may differ in different states
- Represent it using a *state variable*
    - a term that we can assign a value to
        - *e.g.*, loc(r1)
- Let X = {all state variables in the environment}

  e.g., X = {loc(r1), loc(c1), loc(c2), cargo(r1)}
- Each state variable $x \in X$ has a *range*
  = {all values that can be assigned to x}
    - Range(loc(r1)) = *Locs*
    - Range(loc(c1)) = Range(loc(c2)) = *Robots* ∪ *Locs*
    - Range(cargo(r1)) = *Containers* ∪ {nil}
- To abbreviate the "range" notation often I'll just say things like
    - loc(r1) ∈ *Locs*
    - loc(c1), loc(c2) ∈ *Robots* ∪ *Locs*



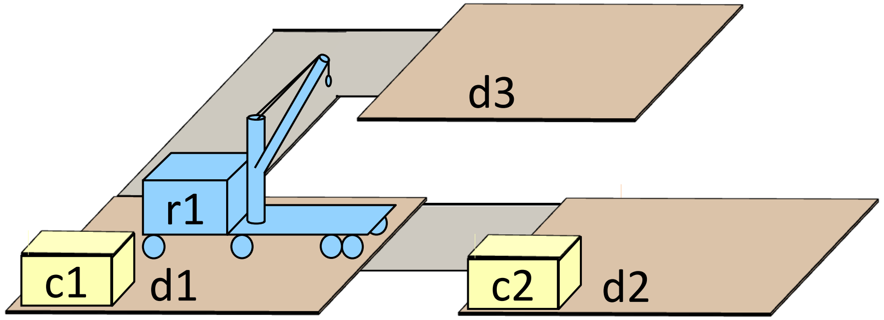
Instead of "domain", to avoid confusion with planning domains

States as Functions

- Represent each state s as a function that assigns values to state variables
  - For each state variable x, $s(x)$ is one x's possible values

s_1(loc(r1)) = d1, s_1(cargo(r1)) = nil,

s_1(loc(c1)) = d1, s_1(loc(c2)) = d2



- Mathematically, a function is a set of ordered pairs

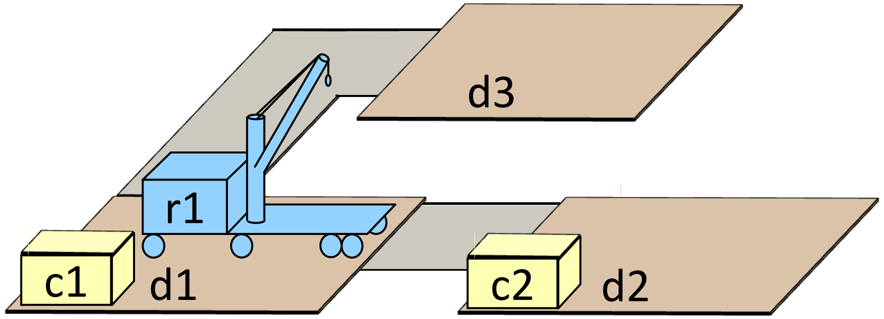
s_1 = {(loc(r1), d1), (cargo(r1), nil), (loc(c1), d1) , (loc(c2), d2)}

- Equivalently, write it as a set of *ground positive literals* (or *ground atoms*):

s_1 = {loc(r1)=d1, cargo(r1)=nil, loc(c1)=d1, loc(c2)=d2}

  - Here, we're using '=' as a predicate symbol

Action Schemas



- Action *schema* (or *template*): parameterized set of actions

  α = (head, pre, eff, cost)
  - head: *name, parameters*
  - pre: *precondition* literals
  - eff: *effect* literals
  - cost: *a number* (optional, default is 1)
- e.g.,
  - head = take(r,l,c)
  - pre = {cargo(r)=nil, loc$(r)=l$, loc$(c)=l$}
  - eff = {cargo$(r)=c$, loc$(c)=r$}
- Each parameter has a range of possible values:
  - Range(r) = *Robots* = {r1}
  - Range(l) = *Locs* = {d1,d2,d3}
  - Range(l) = Range(m) = *Locs* = {d1,d2,d3}
  - Range(c) = *Containers* = {c1,c2}

We'll usually write it more like pseudocode:

```
move($r,l,m$)
    pre: loc($r$)=$l$, adjacent($l,m$)
    eff: loc($r$) $\leftarrow m$

take($r,l,c$)
    pre: cargo($r$)=nil, loc($r$)=$l$, loc($c$)=$l$
    eff: cargo($r$) $\leftarrow c$, loc($c$) $\leftarrow r$

put($r,l,c$)
    pre: loc($r$)=$l$, loc($c$)=$r$
    eff: cargo($r$) $\leftarrow$ nil, loc($c$) $\leftarrow l$

$r \in$ Robots = {r1}
$l,m \in$ Locs = {d1,d2,d3}
$c \in$ Containers = {c1,c2}
```

the *target* of the assignment

Lecture slides for *Acting, Planning, and Learning*. Creative Commons CC BY-SA 4.0

Actions

- $\mathcal{A}$ = set of action schemas

move(r,l,m)
pre: loc(r)=l, adjacent(l, m)
eff: loc(r) ← m

take(r,l,c)
pre: cargo(r)=nil, loc(r)=l, loc(c)=l
eff: cargo(r) ← c, loc(c) ← r

put(r,l,c)
pre: loc(r)=l, loc(c)=r
eff: cargo(r) ← nil, loc(c) ← l

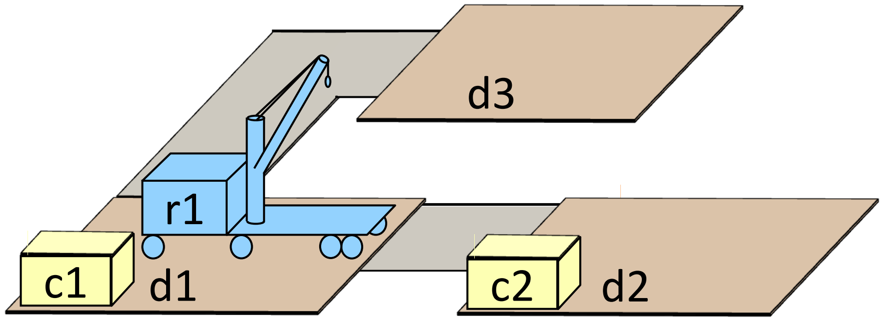
$r \in Robots$ = {r1}

$l,m \in Locs$ = {d1,d2,d3}

$c \in Containers$ = {c1,c2}

- Action: *ground instance* of an $\alpha \in \mathcal{A}$
  - replace each parameter with something in its range
- A = {all actions we can get from $\mathcal{A}$}
  = {all ground instances of members of $\mathcal{A}$}

move(r1,d1,d2)
pre: loc(r1)=d1, adjacent(d1,d2)
eff: loc(r1) ← d2

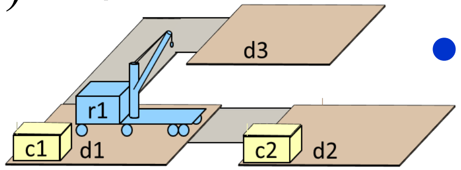


Actions

- $\mathcal{A}$ = set of action schemas

move(r,l,m)
    pre: loc(r)=l, adjacent(l, m)
    eff: loc(r) ← m

take(r,l,c)
    pre: cargo(r)=nil, loc(r)=l, loc(c)=l
    eff: cargo(r) ← c, loc(c) ← r



put(r,l,c)
    pre: loc(r)=l, loc(c)=r
    eff: cargo(r) ← nil, loc(c) ← l

$r \in Robots$ = {r1}
$l,m \in Locs$ = {d1,d2,d3}
$c \in Containers$ = {c1,c2}

- Action: *ground instance* a of an action schema $\alpha \in \mathcal{A}$ such that no state variable is a target of more than one effect eff(a)
- A = {all actions we can derive from $\mathcal{A}$}
  = {all ground instances of members of $\mathcal{A}$}

move(r1,d1,d2)
    pre: loc(r1)=d1, adjacent(d1,d2)
    eff: loc(r1) ← d2

- We'll normally refer to an action by writing its head
  - move(r1,d1,d2)

Poll. Let:
$\mathcal{A}$ = {the action schemas on this page}
A = {all ground instances of members of $\mathcal{A}$}
How many move actions in A?

Answers:
A. 1
B. 2
C. 3
D. 4
E. 5
F. 6
G. 7
H. 8
I. 9
J. other

Lecture slides for *Acting, Planning, and Learning*. Creative Commons CC BY-SA 4.0

Applicability

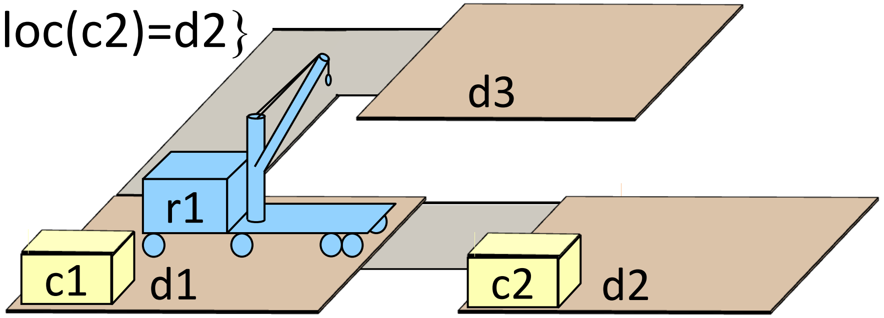
- a is *applicable* in s if
  - for every positive literal $l \in \text{pre}(a)$, $l \in s$ or l is in one of the rigid relations
  - for every negative literal $\neg l \in \text{pre}(a)$, $l \notin s$ and l isn't in any of the rigid relations

- Rigid relation

  adjacent = {(d1,d2), (d2,d1), (d1,d3), (d3,d1)}

- *State*

  s_1 = {loc(r1)=d1, cargo(r1)=nil, loc(c1)=d1, loc(c2)=d2}



- Action schema

  move(r,l,m)
  pre: loc(r)=l, adjacent(l, m)
  eff: loc(r) ← m

  $r \in Robots$ = {r1}

  $l,m \in Locs$ = {d1,d2,d3}

- Applicable:

  move(r1,d1,d2)
  pre: loc(r1)=d1, adjacent(d1,d2)
  eff: loc(r1) ← d2

- Not applicable:

  move(r1,d2,d1)
  pre: loc(r1)=d2, adjacent(d2,d1)
  eff: loc(r1) ← d1

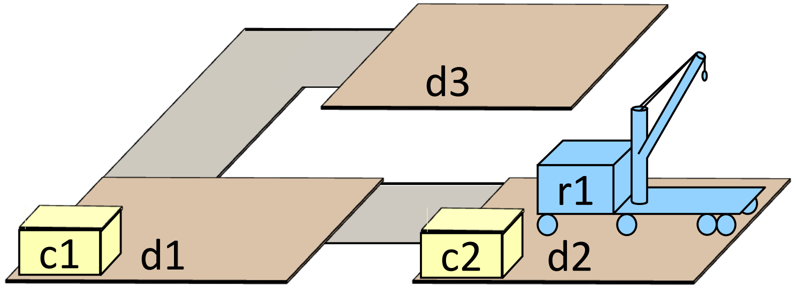
Poll: How many move actions are applicable in s_1?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6
G. 7
H. 8
I. 9
J. other

Applying an Action

- If a is applicable in s:
  - $\gamma(s,a) = \{x = w \mid \text{eff}(a) \text{ contains } x \leftarrow w\}$
    $\cup \; \{x = w \mid x \text{ isn't a target in eff}(a)\}$

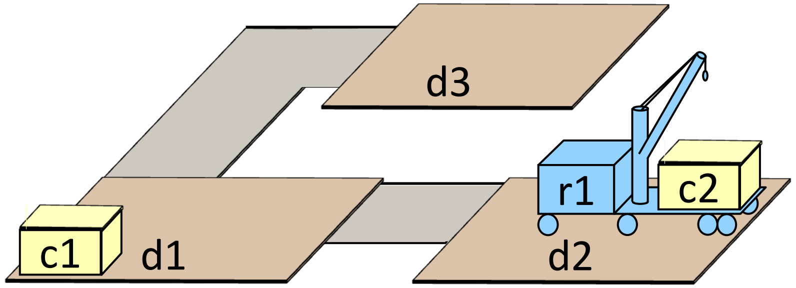
- s_2 = {loc(r1)=d2, cargo(r1)=nil, loc(c1)=d1, loc(c2)=d2}



- a = take(r1,c2,d2)
  pre: cargo(r1)=nil, loc(r1)=d2, loc(c2)=d2
  eff: cargo(r1) ← c2, loc(c2) ← r1

- γ(s_2, take(r1,c2,d2)) =

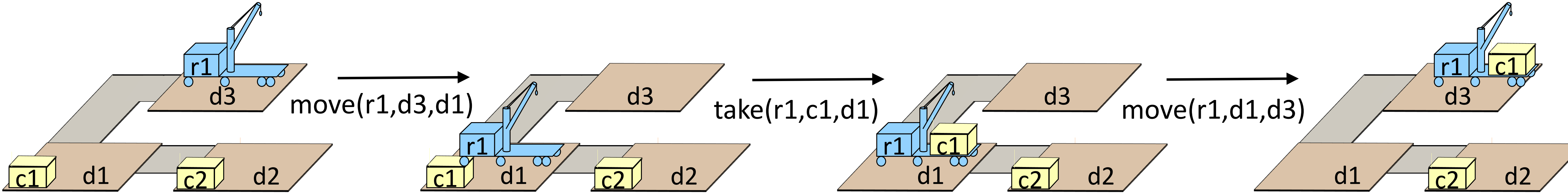
$$\{\underbrace{\text{loc(r1)=d2, loc(c1)=d1,}}_{\text{from } s_2}\ \underbrace{\text{cargo(r1)=c2, loc(c2)=r1}}_{\text{from eff}(a)}\}$$



Applying a Plan

- A plan π is applicable in a state s if we can apply the actions in the order that they appear in π
- This produces a sequence of states
- $\gamma(s,\pi)$ = the last state in the sequence

- $\pi = \langle$move(r1,d3,d1), take(r1,c1,d1), move(r1,d1,d3)$\rangle$
- $\gamma(s_0,\pi) = s_3$
- $\hat{\gamma} = \langle s_0, s_1, s_2, s_3 \rangle$



s_0 = {loc(r1)=d3,
cargo(r1)=nil,
loc(c1)=d1,
loc(c2)=d2}

s_1 = {loc(r1)=d1,
cargo(r1)=nil,
loc(c1)=d1,
loc(c2)=d2}

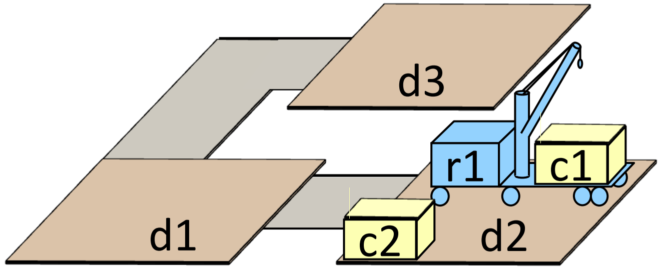
s_2 = {loc(r1)=d1,
cargo(r1)=c1,
loc(c1)=r1,
loc(c2)=d2}

s_3 = {loc(r1)=d3,
cargo(r1)=c1,
loc(c1)=r1,
loc(c2)=d2}

State-Variable Planning Domain

- Let
  - O = ontology of typed objects
  - R = set of rigid relations
  - X = set of lifted state variables, including specifications of their ranges
  - $\mathcal{A}$ = finite set of action schemas
- $(O, R, X, \mathcal{A})$ represents $\Sigma = (S, A, \gamma, \text{cost})$, where
  - A = {all actions induced by $\mathcal{A}$}
  - $\gamma(s,a) = \{x{=}w \mid \text{eff}(a) \text{ contains } x{\leftarrow}w\} \cup \{x{=}w \mid x \text{ isn't a target in eff}(a)\}$
  - cost(.) is as specified in the action schemas
  - S = all states $\{x_1 = v_1, \ldots, x_n = v_n\}$, where
    - $\{x_1, \ldots, x_n\}$ = {all of the ground instances of members of X}
    - each v_i is an object in $Range(x_i)$

s_0 = {loc(r1)=d2,
cargo(r1)=c1,
loc(c1)=r1,
loc(c2)=d2}



O:
- $Objects = Robots \cup Containers \cup Locs \cup \{\text{nil}\}$
- $Robots = \{\text{r1}\}$
- $Containers = \{\text{c1, c2}\}$
- $Locs = \{\text{d1, d2, d3}\}$

R:
- adjacent = {(d1,d2), (d2,d1), (d1,d3), (d3,d1)}

X:
- loc(c) ∈ $Locs \cup Robots$,
- loc(r) ∈ $Locs$,
- cargo(r) ∈ $Containers \cup \{\text{nil}\}$
- where $c \in Containers$, $r \in Robots$

$\mathcal{A}$:

move(r,l,m)
  pre: loc(r)=l, adjacent(l, m)
  eff: loc(r) ← m

take(r,c,l)
  pre: cargo(r)=nil, loc(r)=l, loc(c)=l
  eff: cargo(r) ← c, loc(c) ← r

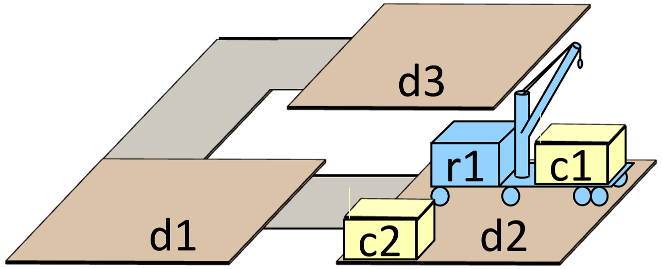
put(r,c,l)
  pre: loc(r)=l, loc(c)=r
  eff: cargo(r) ← nil, loc(c) ← l

Lecture slides for *Acting, Planning, and Learning*. Creative Commons CC BY-SA 4.0

State-Variable Planning Domain

- S = all states $\{x_1 = v_1, \ldots, x_n = v_n\}$, where
  - $\{x_1, \ldots, x_n\}$ = {all of the ground instances of members of $\hat{X}$}
  - each v_i is an object in $Range(\hat{x}_i)$
- S may contain some nonsensical states
  - e.g., states in which both loc(c1)=r1 and cargo(r1)=nil
- But if s_0 and $\mathcal{A}$ are defined properly, applying a plan in s_0 will never generate a nonsensical state

s_0 = {loc(r1)=d2,
cargo(r1)=c1,
loc(c1)=r1,
loc(c2)=d2}



O:
- $Objects = Robots \cup Containers \cup Locs \cup \{\text{nil}\}$
- $Robots = \{\text{r1}\}$
- $Containers = \{\text{c1, c2}\}$
- $Locs = \{\text{d1, d2, d3}\}$

R:
- adjacent = {(d1,d2), (d2,d1), (d1,d3), (d3,d1)}

X:
- loc(c) ∈ $Locs \cup Robots$,
- loc(r) ∈ $Locs$,
- cargo(r) ∈ $Containers \cup \{\text{nil}\}$
- where $c \in Containers$, $r \in Robots$

$\mathcal{A}$:

move(r,l,m)
pre: loc(r)=l, adjacent(l, m)
eff: loc(r) ← m

take(r,c,l)
pre: cargo(r)=nil,
loc(r)=l, loc(c)=l
eff: cargo(r) ← c, loc(c) ← r

put(r,c,l)
pre: loc(r)=l, loc(c)=r
eff: cargo(r) ← nil, loc(c) ← l

State-Variable Planning Problem

- $P = (\Sigma, s_0, g)$, where
  - Σ = is a state-variable planning domain
  - $s_0 \in S$ is the initial state
  - g is a set of ground literals called the *goal*

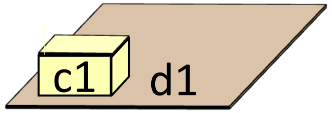
- S_g = {all states in S that satisfy g}
  = {$s \in S \mid s \cup R$ contains every positive literal in g, and none of the negative literals in g}

- π is a *solution* for P if $\gamma(s_0,\pi)$ satisfies g

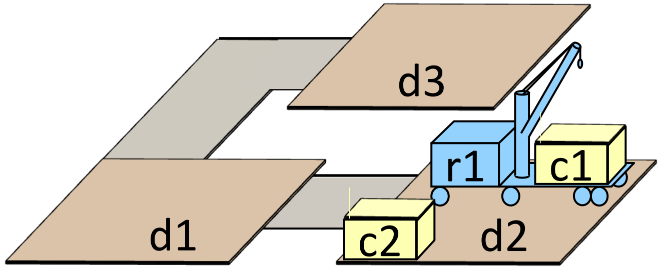
Poll: How many solutions of length 3?

A. 1 B. 2 C. 3
D. 4 E. 5 F. 6
G. 7 H. 8 I. 9
J. other

g = {loc(c1)=d1}



s_0 = {loc(r1)=d2,
cargo(r1)=c1,
loc(c1)=r1,
loc(c2)=d2}



⟨move(r1,d2,d1), put(r1,c1,d1)⟩ is a solution of length 2

O:
Objects = *Robots* ∪ *Containers* ∪ *Locs* ∪ {nil}
Robots = {r1}
Containers = {c1, c2}
Locs = {d1, d2, d3}

R:
adjacent = {(d1,d2), (d2,d1), (d1,d3), (d3,d1)}

X:
loc(c) ∈ *Locs* ∪ *Robots*,
loc(r) ∈ *Locs*,
cargo(r) ∈ *Containers* ∪ {nil}
where c ∈ *Containers*, r ∈ *Robots*

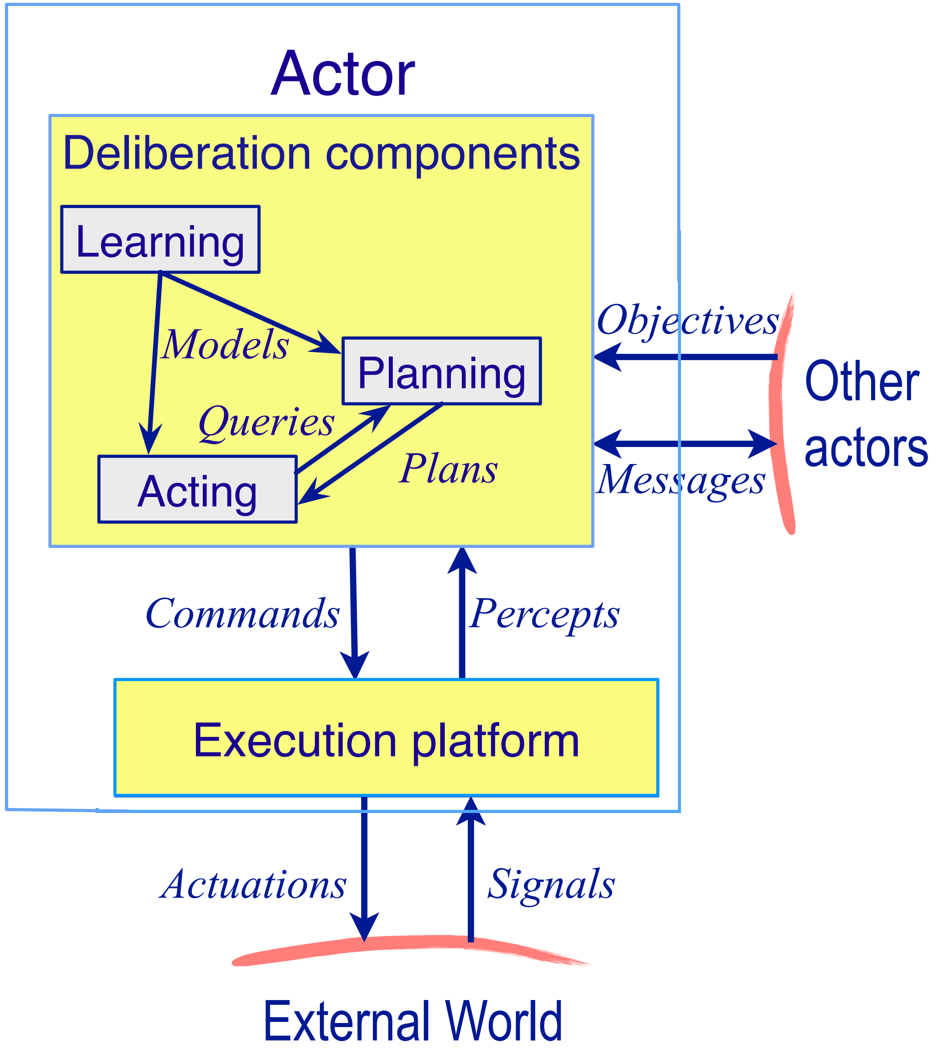
$\mathcal{A}$:
move(r,l,m)
  pre: loc(r)=l, adjacent(l, m)
  eff: loc(r) ← m
take(r,c,l)
  pre: cargo(r)=nil, loc(r)=l, loc(c)=l
  eff: cargo(r) ← c, loc(c) ← r
put(r,c,l)
  pre: loc(r)=l, loc(c)=r
  eff: cargo(r) ← nil, loc(c) ← l

Lecture slides for *Acting, Planning, and Learning*. Creative Commons CC BY-SA 4.0

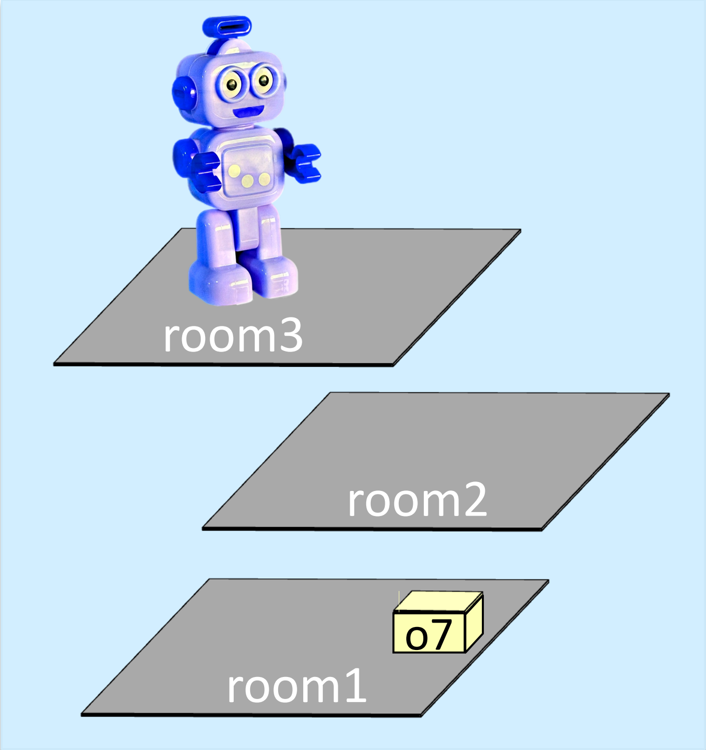
Section 2.3. Acting

- For classical planning problems we assumed
    - Finite, static world, just one actor
    - No concurrent actions, no explicit time
    - Determinism, no uncertainty, no exogeneous events
    - Full observability
    - Unit-cost actions
  - Sequence of states and actions $\langle s_0, a_1, s_1, a_2, s_2, \ldots \rangle$
- Most real-world environments don't satisfy the assumptions because of errors in prediction
- This can usually be fine if
  - errors occur infrequently, and
  - they don't have severe consequences

- What to do if an error *does* occur?



Service Robot



$\pi = \langle a_1, a_2, a_3, a_4, a_5 \rangle$

a_1 = go(r1,room3,hall)
a_2 = navigate(r1,hall,room1)
a_3 = take(r1,o7,room1)
a_4 = navigate(r1,room1,room2)
a_5 = put(r1,o7,room2)⟩

go(r,l,m)
pre: adjacent(l,m), loc(r)=l
eff: loc(r) ← m

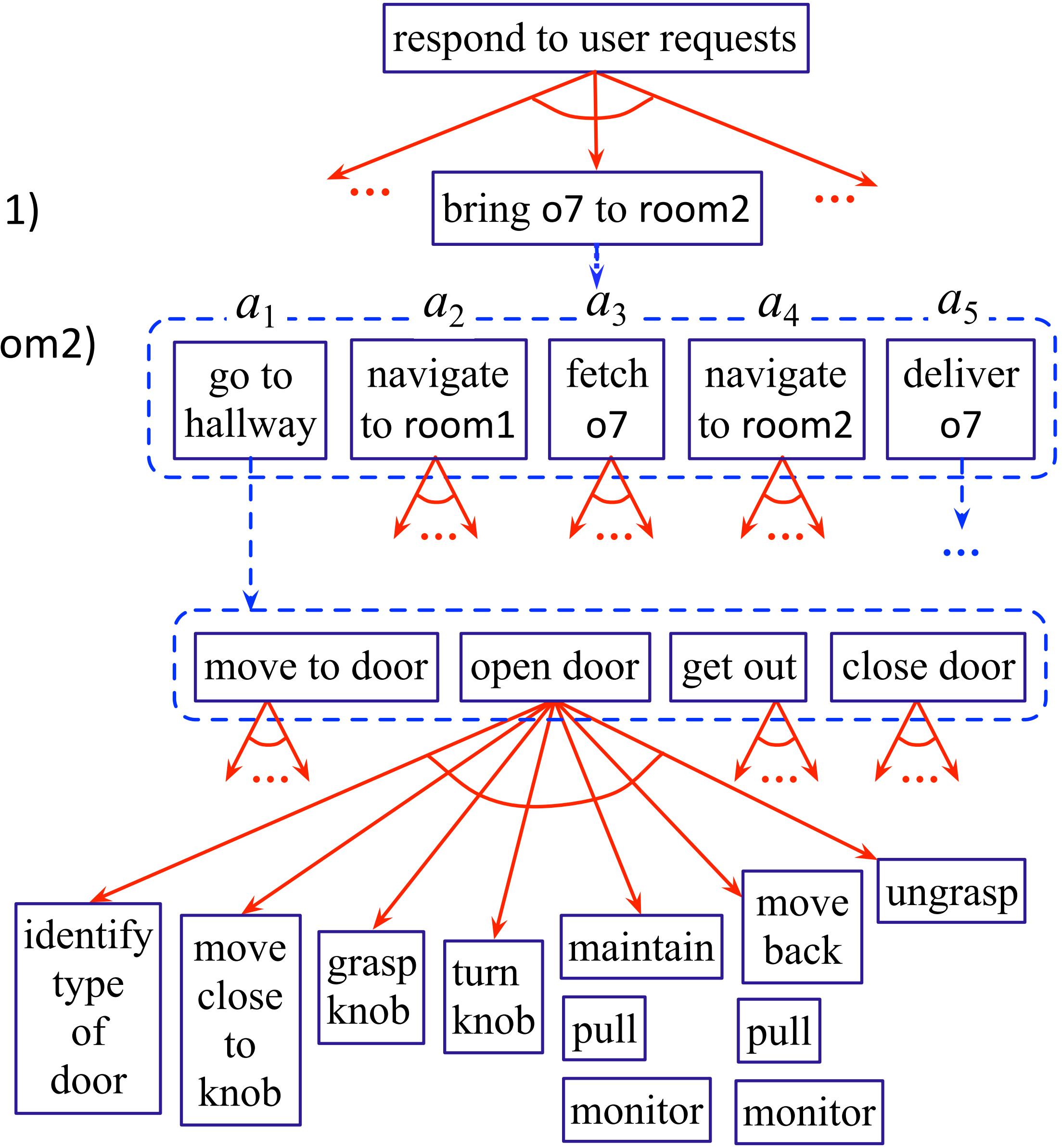
ignores how to get from l to m, e.g., opening the door

navigate(r,l,m)
pre: ¬adjacent(l, m), loc(r)=l
eff: loc(r) ← m

ignores how do navigation, localization

take(r,o,l)
pre: loc(r)=l, loc(o)=l,
cargo(r)=nil
eff: loc(o) ← r, cargo(r) ← o

ignores how to grasp o, lift it, put it down


respond to user requests

...
bring o7 to room2
...

a_1 a_2 a_3 a_4 a_5

go to hallway
navigate to room1
fetch o7
navigate to room2
deliver o7

... ...

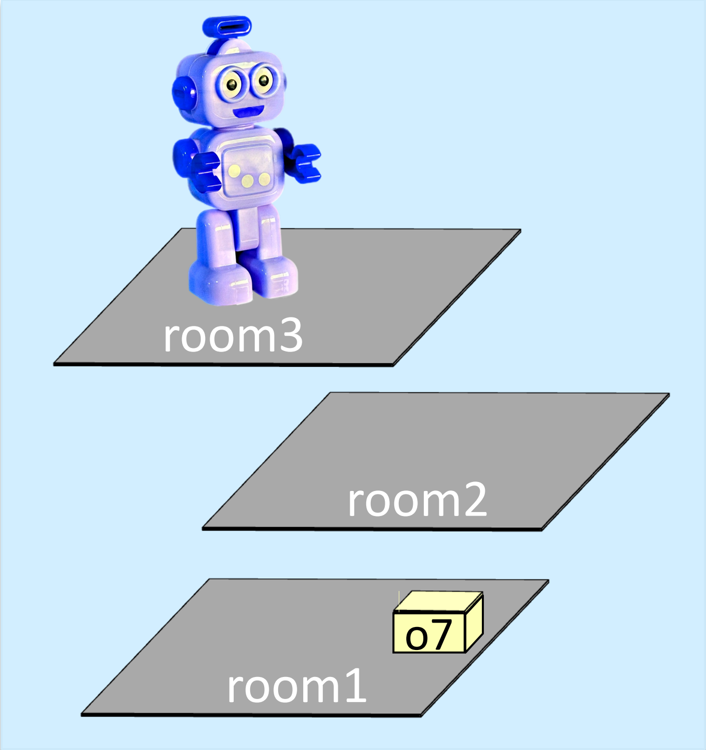
move to door
open door
get out
close door

... ...

identify type of door
move close to knob
grasp knob
turn knob
maintain
pull
monitor
move back
pull
monitor
ungrasp

Lecture slides for *Acting, Planning, and Learning*. Creative Commons CC BY-SA 4.0

Service Robot



go(r,l,m)
    pre: adjacent(l,m), loc(r)=l
    eff: loc(r) $\leftarrow m$

navigate(r,l,m)
    pre: ¬adjacent(l, m), loc(r)=l
    eff: loc(r) $\leftarrow m$

take(r,o,l)
    pre: loc(r)=l, loc(o)=l,
        cargo(r)=nil
    eff: loc(o) $\leftarrow r$, cargo(r) $\leftarrow o$

$\pi = \langle a_1, a_2, a_3, a_4, a_5 \rangle$

a_1 = go(r1,room3,hall)

a_2 = navigate(r1,hall,room1)

a_3 = take(r1,o7,room1)

a_4 = navigate(r1,room1,room2)

a_5 = put(r1,o7,room2)⟩

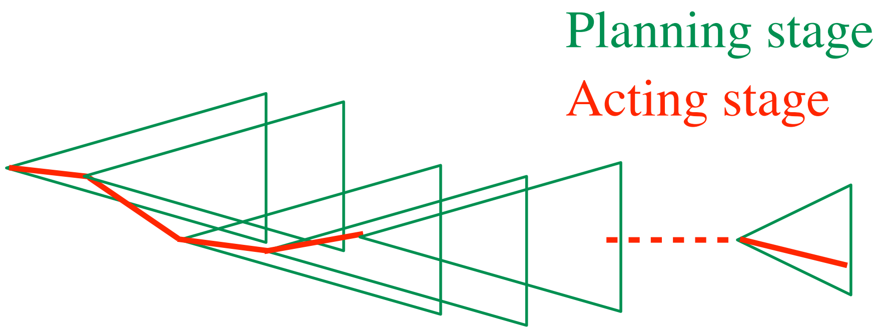
- Some things that can go wrong:
  - *Execution failures*
    - robot gripper slips on doorknob
    - door is locked or broken
  - *Sensor errors*
    - navigation error causes robot to go to wrong room
  - *Incorrect or partial information*
    - where is o7 ?
  - *Events that make actions inapplicable*
    - someone puts object o6 onto r1
  - *Events that make actions unnecessary*
    - someone puts object o7 onto r1

- How to detect and recover?

Acting with Lookahead

```
Run-Lookahead($\Sigma, g$)
    $s \leftarrow$ abstraction of observed state $\xi$
    while $s \not\models g$ do
        $\pi \leftarrow$ *Lookahead*($\Sigma, s, g$)
        if $\pi$ = failure then return failure
        $a \leftarrow$ pop-first-action($\pi$); perform($a$)
        $s \leftarrow$ abstraction of observed state $\xi$
```

the planner

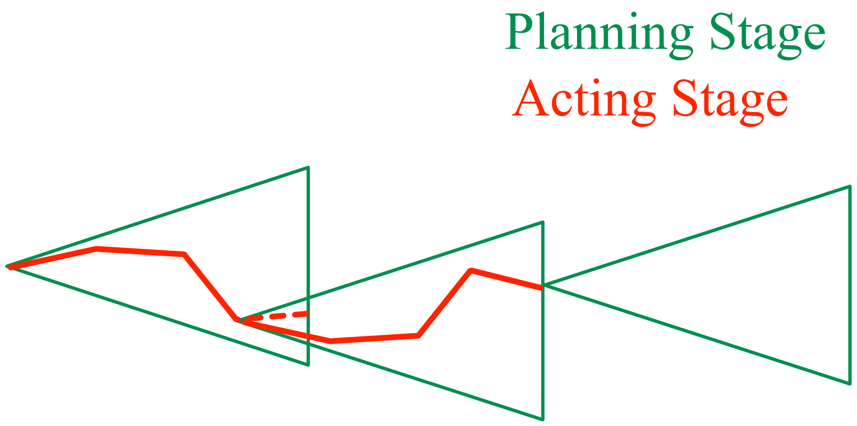


- Call *Lookahead*, obtain π, perform 1st action, call *Lookahead* again …
- Useful when unpredictable things are likely to happen
  - Replans immediately
- Also useful with *receding horizon* search (e.g., as in chess programs):
  - Lookahead looks a limited distance ahead
- Potential problem:
  - Lookahead needs to return quickly
  - Otherwise, may pause repeatedly while waiting for Lookahead to return
  - What if ξ changes during the wait?

Lecture slides for *Acting, Planning, and Learning*. Creative Commons CC BY-SA 4.0

Acting with Lookahead

```
Run-Lazy-Lookahead($\Sigma, g$)
    $\pi \leftarrow \langle\,\rangle$
    while True do
        $s \leftarrow$ abstraction of observed state $\xi$
        if $s \vDash g$ then return success
        if $\pi = \langle\,\rangle$ or Simulate($\Sigma, s, g, \pi$) = failure then
            $\pi \leftarrow$ Lookahead($\Sigma, s, g$)
            if $\pi$ = failure then return failure
        $a \leftarrow$ pop-first-action($\pi$)
        perform($a$)
```



- Call *Lookahead*, execute the plan as far as possible, don't call *Lookahead* again unless necessary
- *Simulate* tests whether the plan will execute correctly
  - Could do lower-level refinement, physics-based simulation
  - Could just test whether $\gamma(s,\pi) \vDash g$
  - Or just test whether $s = \gamma(s', a)$, where s' is the previous state
- Potential problems
  - *Simulate* needs to return quickly
    - otherwise, may pause repeatedly, ξ may change
  - May might miss opportunities to replace π with a better plan

Poll: Assuming no action failures during acting, which approach does more work, in terms of planning: Run-Lazy-Lookahead or Run-Lookahead?

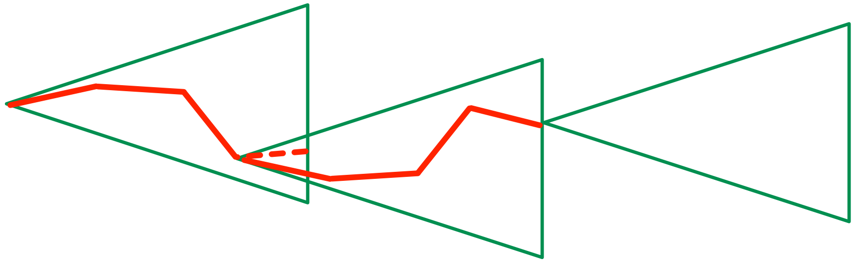
A. Run-Lazy-Lookahead	C. Equal amounts
B. Run-Lookahead	D. Unsure

Lecture slides for *Acting, Planning, and Learning*. Creative Commons CC BY-SA 4.0

Acting with Plan Repair

- We may want to repair π rather than get a new plan
    - e.g., if we've already made commitments or resource allocations
- Modify Run-Lazy-Lookahead

```
Run-Lazy-Lookahead($\Sigma, g$)
    $\pi \leftarrow \langle\,\rangle$
    while True do
        $s \leftarrow$ abstraction of observed state $\xi$
        if $s \models g$ then return success
        if $\pi = \langle\,\rangle$ or Simulate($\Sigma, s, g, \pi$) = failure then
            $\pi \leftarrow$ Lookahead-Repair($\Sigma, s, g, \pi$)
            if $\pi$ = failure then return failure
        $a \leftarrow$ pop-first-action($\pi$)
        perform($a$)
```

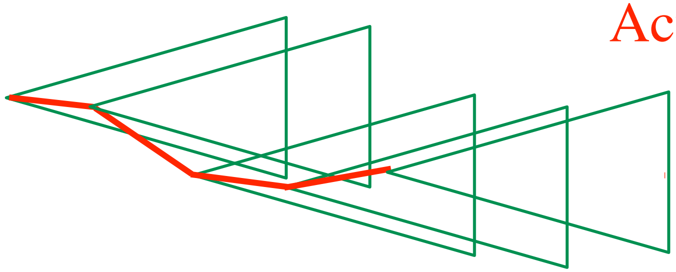


Planning Stage
Acting Stage

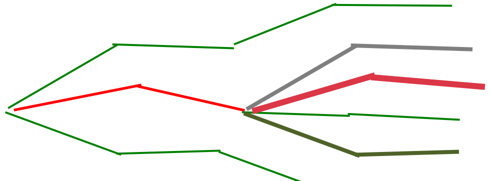
How to do Lookahead

Some possibilities (can also combine these)

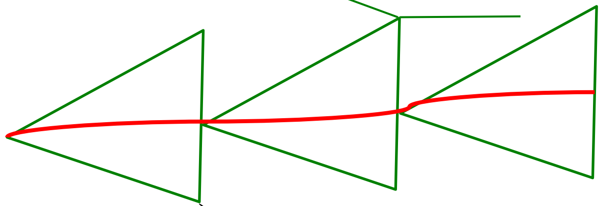
- **Full planning** (if the planner can solve the planning problem quickly enough)
- **Receding horizon**
  - Modify Lookahead to search just part of the way to g
  - E.g., cut off search when one of the following exceeds a maximum threshold:
    - plan length, plan cost, computation time



- **Sampling**
  - Modify Lookahead to do a *Monte Carlo rollout*
    - Depth-first search with random node selection and no backtracking
  - Call Lookahead several times, choose the plan that looks best
  - Best-known example of this: the UCT algorithm (see Chapter 9)



- **Subgoaling**
  - Tell Lookahead to plan for some subgoal g_1, rather than g itself (see next page)
  - Once the actor has achieved g_1, tell Lookahead to plan for the next subgoal g_2
  - And so forth until the actor reaches g



Subgoaling Example

- **Killzone 2**
  - “First-person shooter” game, ≈ 2009
  - widely acclaimed at the time
- Special-purpose AI planner
  - Plans enemy actions at the squad level
    - Subproblems; plans are maybe 4–6 actions long
  - Different planning algorithm from what we’ve discussed so far
      - HTN planning (see Part II)
    - Quickly generates a plan for a subgoal
    - Replans several times per second as the world changes
- Why it worked:
  - Don’t *want* to get the best possible plan
  - Need actions that appear believable and consistent to human users
  - Need them very quickly

Classical Representation

- Motivation
  - The field of AI planning started out as automated theorem proving
  - It still uses a lot of that notation
- Classical representation is equivalent to state-variable representation
  - No distinction between rigid and varying properties
  - Both represented as logical predicates
  - Both are in the current state

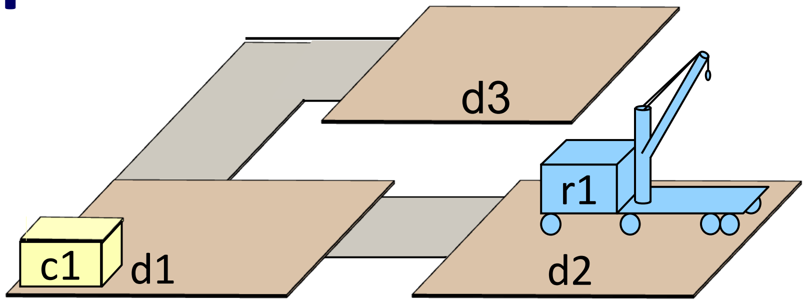
adjacent(l,m) - location l is adjacent to m

loc(r) = l ⟶ loc(r,l) - robot r is at location l

loc(c) = r ⟶ loc(c,r) - container c is on robot r

cargo(r) = c ⟶ loaded(r) - there's a container on r

why not loaded(r,c)?



- State s = a set of ground atoms
  - Atom a is true in s iff $a \in s$

s_0 = {adjacent(d1,d2), adjacent(d2,d1),
adjacent(d1,d3), adjacent(d3,d1),
loc(c1,d1), loc(r1,d2)}

Poll: Should s_0 also contain $\neg$ loaded(r1) ?

A: yes B: no

C: unsure

Lecture slides for *Acting, Planning, and Learning*. Creative Commons CC BY-SA 4.0

Classical planning operators

- action schemas

move(r,l,m)
  pre: loc$(r)=l$, adjacent(l, m)
  eff: loc$(r) \leftarrow m$

take(r,c,l)
  pre: cargo(r)=nil, loc$(r)=l$, loc(c)$=l$
  eff: cargo$(r) \leftarrow c$, loc(c) $\leftarrow r$

put(r,c,l)
  pre: loc$(r)=l$, loc$(c)=r$
  eff: cargo$(r) \leftarrow$ nil, loc(c) $\leftarrow l$

Range(r) = *Robots* = {r1}

Range(l) = Range(m) = *Locs* = {d1,d2,d3}

Range(c) = *Containers* = {c1,c2}

- Classical planning operators

move(r,l,m)
  pre: loc(r,l), adjacent(l, m)
  eff: ¬loc(r,l), loc(r,m)

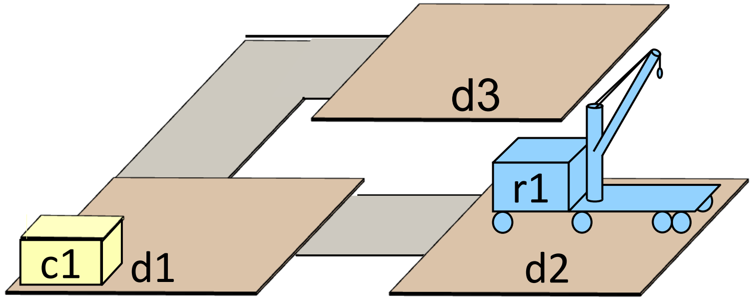
take(r,c,l)
  pre: ¬loaded(r), loc(r,l), loc(c,l)
  eff: loaded(r), ¬loc(c,l), loc(c,r)

put(r,c,l)
  pre: loc(r,l), loc(c,r)
  eff: ¬loaded(r), loc(c,l), ¬loc(c,r)

Poll: Does move really need to include ¬loc(r,l) ?

A: yes B: no

C: unsure



Lecture slides for *Acting, Planning, and Learning*. Creative Commons CC BY-SA 4.0

Classical Actions

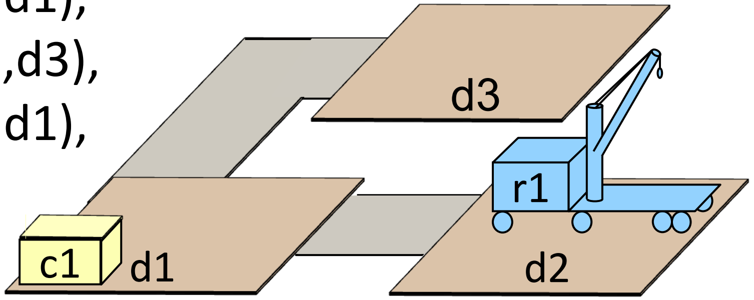
- Planning operator:

o: move(r,l,m)
pre: loc(r,l), adjacent(l,m)
eff: ¬loc(r,l), loc(r,m)

- Action:

a_1: move(r1,d2,d1)
pre: loc(r1,d2), adjacent(d2,d1)
eff: ¬loc(r1,d2), loc(r1,d1)

s_0 = {adjacent(d1,d2),
adjacent(d2,d1),
adjacent(d1,d3),
adjacent(d3,d1),
loc(c1,d1),
loc(r1,d2)}



- Let
  - $\text{pre}^-(a)$ = {a's negated preconditions}
  - $\text{pre}^+(a)$ = {a's non-negated preconditions}

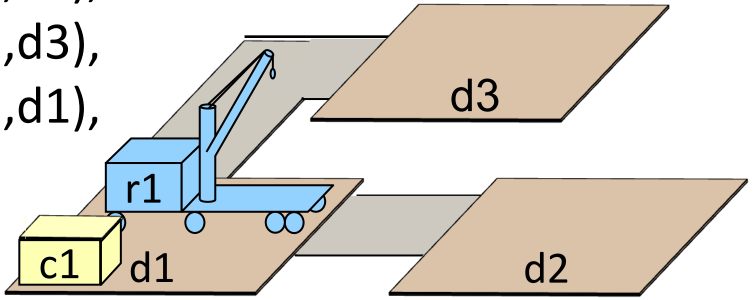
- a is applicable in state s iff

$s \cap \text{pre}^-(a) = \emptyset$ and $\text{pre}^+(a) \subseteq s$

- If a is applicable in s then
  - $\gamma(s,a) = (s \setminus \text{eff}^-(a)) \cup \text{eff}^+(a)$

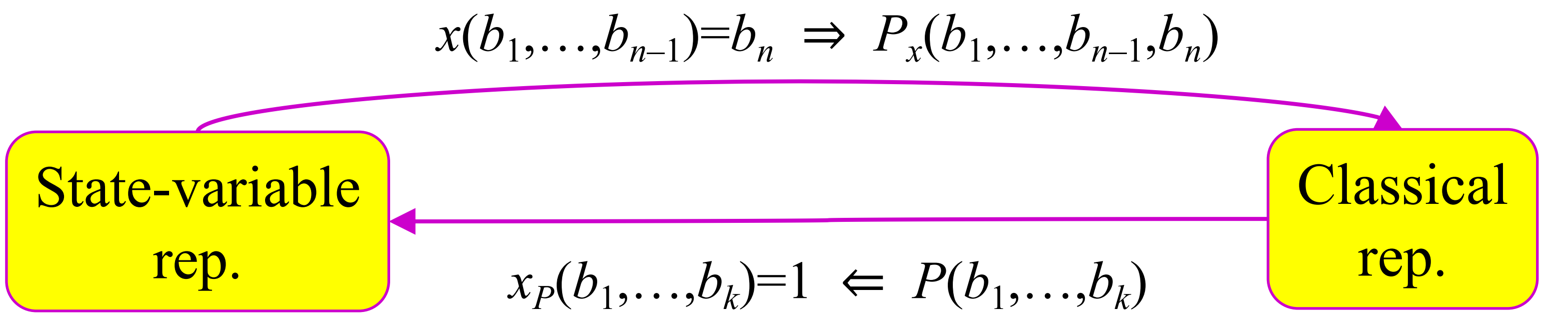
meaning?

$\gamma(s_0, a_1)$ = {adjacent(d1,d2),
adjacent(d2,d1),
adjacent(d1,d3),
adjacent(d3,d1),
loc(c1,d1),
loc(r1,d1)}



Lecture slides for *Acting, Planning, and Learning*. Creative Commons CC BY-SA 4.0

Discussion



$x(b_1,\ldots,b_{n-1})=b_n \;\Rightarrow\; P_x(b_1,\ldots,b_{n-1},b_n)$

$x_P(b_1,\ldots,b_k)=1 \;\Leftarrow\; P(b_1,\ldots,b_k)$

- Equivalent expressive power
  - Each can be converted to the other in linear time and space
- Classical representation
  - More natural for logicians
  - Don't require single-valued functions
- State variables
  - More natural for engineers and computer programmers
  - When changing a value, don't have to explicitly delete the old one
- Historically, classical representation has been more widely used
  - That's starting to change

Poll: Could we instead use $x_P(b_1,\ldots,b_{k-1})=b_k$?

A: yes B: no

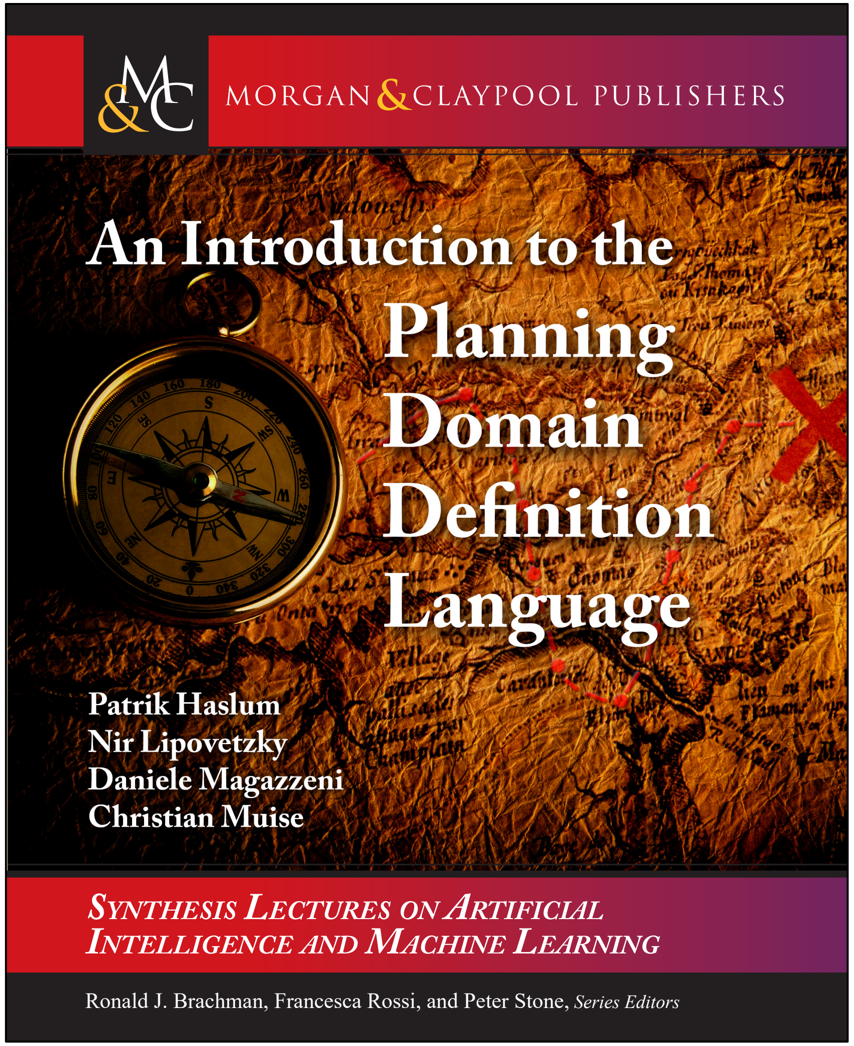
C: unsure

Lecture slides for *Acting, Planning, and Learning*. Creative Commons CC BY-SA 4.0

PDDL

- Language for defining planning domains and problems
- Original version of PDDL ≈ 1996
  - Just classical planning
- Multiple revisions and extensions
  - Different subsets accommodate different kinds of planning

- We'll discuss the classical-planning subset
  - Chapter 2 of the PDDL book


MORGAN & CLAYPOOL PUBLISHERS

An Introduction to the Planning Domain Definition Language

Patrik Haslum
Nir Lipovetzky
Daniele Magazzeni
Christian Muise

Synthesis Lectures on Artificial Intelligence and Machine Learning

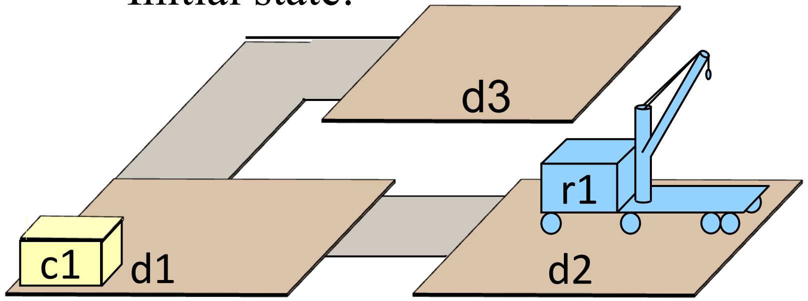
Ronald J. Brachman, Francesca Rossi, and Peter Stone, *Series Editors*

Example domain

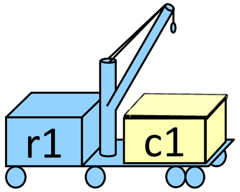
```
(define (domain example-domain-1)
    (requirements :negative-preconditions)

    (:action   move
     :parameters (?r ?l ?m)
     :precondition (and (loc ?r ?l)
                        (adjacent ?l ?m))
     :effect (and (not (loc ?r ?l))
                  (loc ?r ?m)))

    (:action take
     :parameters (?r ?l ?c)
     :precondition (and (loc ?r ?l)
                        (loc ?c ?l)
                        (not (loaded ?r)))
     :effect (and (not (loc ?c ?l))
                  (loc ?c ?r)
                  (loaded ?r)))

    (:action put
     :parameters (?r ?l ?c)
     :precondition (and (loc ?r ?l)
                       (loc ?c ?r))
     :effect (and (loc ?c ?l)
                  (not (loc ?c ?r))
                  (not (loaded ?r)))))
```

These are untyped parameters.

Initial state:



Goal:

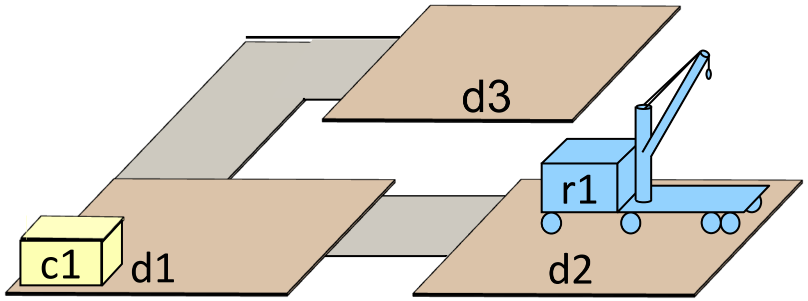


```
(define (problem example-problem-1)
    (:domain example-domain-1))

    (:init
      (adjacent d1 d2)
      (adjacent d2 d1)
      (adjacent d1 d3)
      (adjacent d3 d1)
      (loc c1 d1)
      (loc r1 d2)

    (:goal (loc c1 r1)))
```

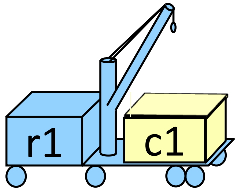
Lecture slides for *Acting, Planning, and Learning*. Creative Commons CC BY-SA 4.0

Example problem

- Classical representation:



s_0 = {adjacent(d1,d2), adjacent(d2,d1),
adjacent(d1,d3), adjacent(d3,d1),
loc(c1,d1), loc(r1,d2)}



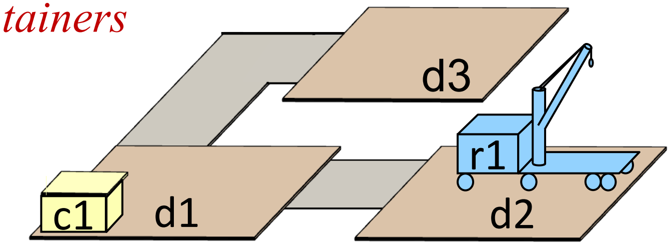
g = {loc(c1,r1)}

```
(define (problem example-problem-1)
    (:domain example-domain-1))

    (:init
      (adjacent d1 d2)
      (adjacent d2 d1)
      (adjacent d1 d3)
      (adjacent d3 d1)
      (loc c1 d1)
      (loc r1 d2)

    (:goal (loc c1 r1)))
```

Typed domain

State-variable representation:

- *Objects* = *Movable_objects* ∪ *Locs*
- *Movable_objects* = *Robots* ∪ *Containers*
- *Robots* = {r1}
- *Containers* = {c1}
- *Locs* = {d1, d2, d3}



- *r* ∈ *Robots*, *l*,*m* ∈ *Locs*, *c* ∈ *Containers*

```
(define (domain example-domain-2)
    (:requirements
        :negative-preconditions
        :typing)
    (:types
        location movable-obj - object
        robot container - movable-obj)

    (:predicates
        (loc ?r - movable-obj
             ?l - location)
        (loaded ?r - robot)
        (adjacent ?l ?m - location))
```

Locations, Movable_objects ⊆ *Objects*

Robots, Containers ⊆ *Movable_objects*

r ∈ *Movable_objects*

l ∈ *Locs*

r ∈ *Robots*

```
(:action move
 :parameters (?r - robot
              ?l ?m - location)
 :precondition (and (loc ?r ?l)
                    (adjacent ?l ?m))
 :effect (and (not (loc ?r ?l))
              (loc ?r ?m)))

(:action take
 :parameters (?r - robot
              ?l - location
              ?c - container)
 :precondition (and (loc ?r ?l)
                    (loc ?c ?l)
                    (not (loaded ?r)))
 :effect (and (not (loc ?r ?l))
              (loc ?r ?m)))

(:action put
 :parameters (?r - robot
              ?l - location
              ?c - container)
 :precondition (and (loc ?r ?l)
                    (loc ?c ?r))
 :effect (and (loc ?c ?l)
              (not (loc ?c ?r))
              (not (loaded ?r)))))
```

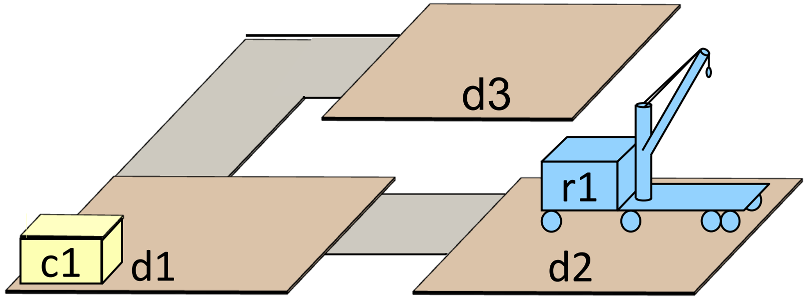
r ∈ *Robots*,
l ∈ *Locs*,
c ∈ *Containers*

Lecture slides for Acting, Planning, and Learning. Creative Commons CC BY-SA 4.0

Typed problem

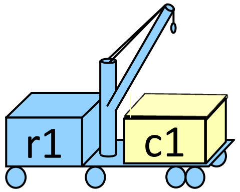
State-variable representation:

  - $Objects = Movable\_objects \cup Locs$
  - $Movable\_objects = Robots \cup Containers$
  - $Robots = \{\text{r1}\}$
  - $Containers = \{\text{c1}\}$
  - $Locs = \{\text{d1, d2, d3}\}$
- $r \in Robots$,
  $l,m \in Locs$,
  $c \in Containers$



$s_0 = \{$adjacent(d1,d2), adjacent(d2,d1),
adjacent(d1,d3), adjacent(d3,d1),
loc(c1,d1), loc(r1,d2)$\}$

$g = \{$loc(c1,r1)$\}$

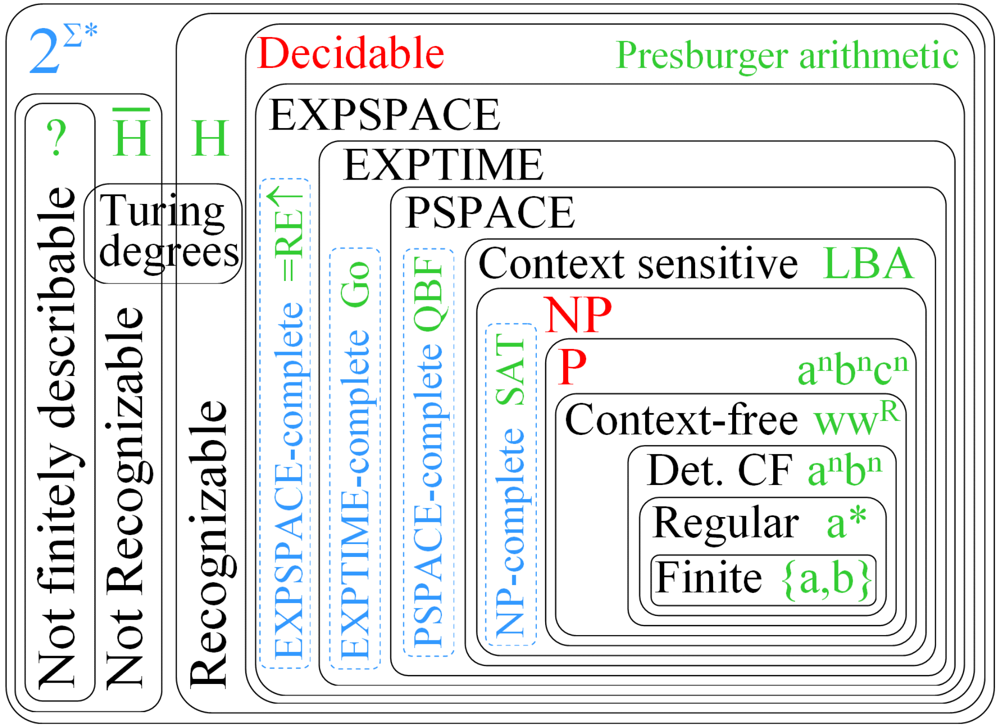


```
(define (problem example-problem-2)
    (:domain example-domain-2))

    (:objects
        r1 - robot
        c1 - container
        d1 d2 d3 - location)

    (:init
      (adjacent d1 d2)
      (adjacent d2 d1)
      (adjacent d1 d3)
      (adjacent d3 d1)
      (loc c1 d1)
      (loc r1 d2)

    (:goal (loc c1 r1)))
```

Computational Complexity Refresher

- Computational complexity results are normally given for *decision problems*
  - each decision problem is an infinite set of questions with *yes/no* answers
    Two decision problems in which *P* may be any classical planning problem:
  - PLAN EXISTENCE: does *P* have a solution?
  - PLAN LENGTH: does *P* have a solution of length ≤ *k*?

The Extended Chomsky Hierarchy



2^{Σ^*}

Not finitely describable ?

Not Recognizable $\overline{H}$

Turing degrees

Recognizable H

Decidable Presburger arithmetic

EXPSPACE

EXPSPACE-complete =RE↑

EXPTIME

EXPTIME-complete Go

PSPACE

PSPACE-complete QBF

Context sensitive LBA

NP

NP-complete SAT

P $a^nb^nc^n$

Context-free ww^R

Det. CF a^nb^n

Regular a^*

Finite {a,b}

Prof. Gabriel Robins, UVA
https://www.cs.virginia.edu/~robins/cs6160/
Lectures 19-21 cover the key concepts

Section 2.5. Computational Complexity

- Suppose P is given in state-variable representation (rather than enumerating S and A explicitly):
  - PLAN EXISTENCE is EXPSPACE-complete
  - PLAN LENGTH is NEXPTIME-complete

- If we restrict P to be in a fixed planning domain Σ that is known in advance :
  - Both problems are in PSPACE
  - PSPACE-complete for some planning domains

- These are *worst-case* results, average case is often much lower (e.g., polynomial)

Poll. What is the complexity of PLAN EXISTENCE if P is given by enumerating S and A explicitly?

A. PSPACE-complete
B. NP-complete
C. Polynomial
D. something else

As a reminder: P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXPTIME $\subseteq$ NEXPTIME $\subseteq$ EXPSPACE

Need a refresher on complexity? See:
- UVA CS 4102 PSPACE and beyond (Bloomfield, 2011)
- MIT OpenCourseWare 6.006 Computational Complexity Lecture
- MIT OpenCourseWare 6.045 Course, specifically lectures 12, 15, & 16
- UVA CS 6160 (Robins, 2022)

Summary

- Section 2.2. State-transition systems
  - Classical planning assumptions
  - States, actions, transition function
  - Plans, planning problems, solutions
  - Run-Plan

- Section 2.3. State-Variable Representation
  - Objects, rigid properties
  - Varying properties, state variables, states
  - Action schemas, actions, applicability, γ
  - Plans, problems, solutions

- Section 2.4. Classical Representation

- Section 2.5. Computational Complexity

- Section 2.6. Acting
  - Things that can go wrong while acting
  - Run-Lookahead, Run-Lazy-Lookahead
  - Plan repair
  - Interacting with an online planner
    - subgoaling, limited horizon, sampling

- Chapter 2 of Haslum *et al*. (2019)
  - Classical fragment of PDDL
  - Planning domains, planning problems
  - untyped, typed