

# tema 6

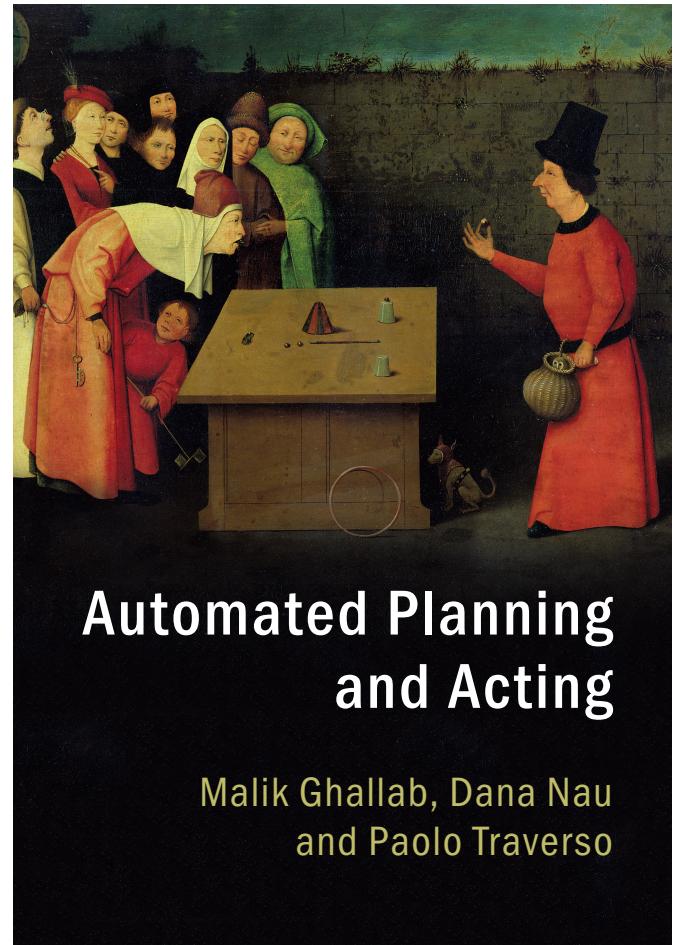
## Chapter 2

### Deliberation with Deterministic Models

**2.4: Backward Search**

**2.5: Plan-Space Search**

Dana S. Nau  
University of Maryland



# Outline

Chapter 2, part *a* (chap2a.pdf):

- 2.1 State-variable representation
    - Comparison with PDDL
  - 2.2 Forward state-space search
  - 2.6 Incorporating planning into an actor
- 

Chapter 2, part *b* (chap2b.pdf):

- 2.3 Heuristic functions
  - 2.7.7 HTN planning
- 

Chapter 2, part *c* (chap2c.pdf):

- Next* → 2.4 Backward search
- 2.5 Plan-space search
- 

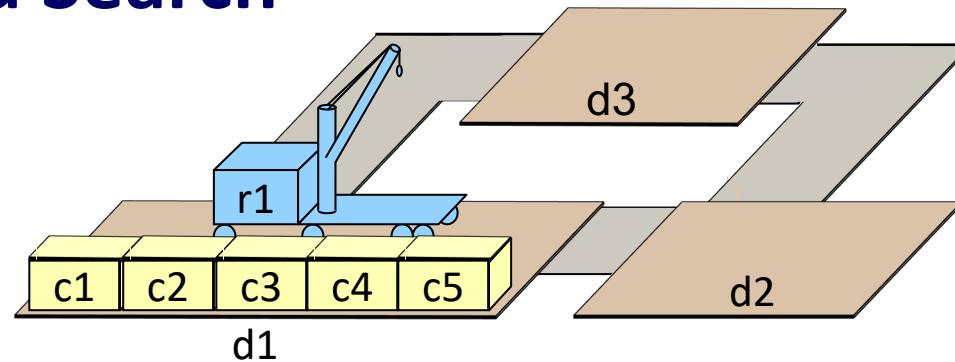
Additional slides:

- 2.7.8 LTL\_planning.pdf

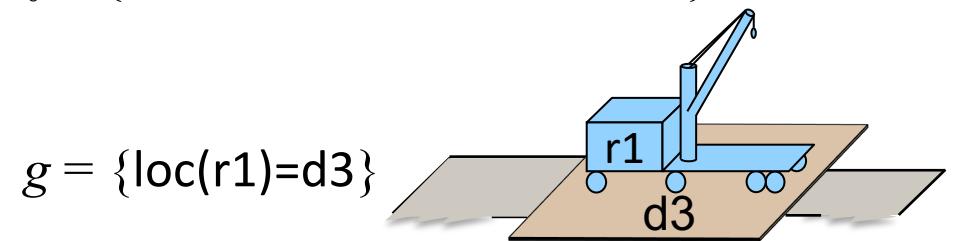


## 2.4 Backward Search

- Forward search: forward from initial state
  - ▶ In state  $s$ , choose applicable action  $a$
  - ▶ Compute state transition  $s' = \gamma(s,a)$
- Backward search: backward from the goal
  - ▶ For goal  $g$ , choose *relevant* action  $a$ 
    - A possible “last action” before the goal
    - Sometimes this has a lower branching factor
- Compute *inverse* state transition  $g' = \gamma^{-1}(g,a)$ 
  - ▶  $g'$  = properties a state  $s'$  should satisfy in order for  $\gamma(s',a)$  to satisfy  $g$
- Equivalently, if  $S_g = \{\text{all states that satisfy } g\}$  then
  - ▶  $S_{g'} = \{\text{all states } s \text{ such that } \gamma(s,a) \in S_g\}$



$$s_0 = \{\text{loc}(c1)=d1, \text{loc}(c2)=d1, \dots\}$$



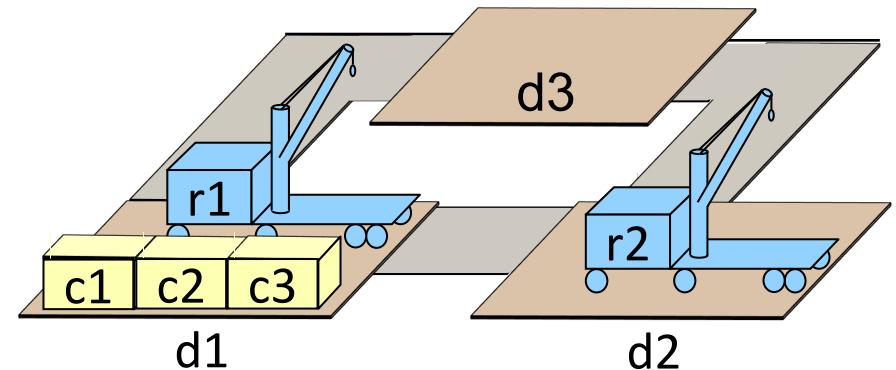
$$g = \{\text{loc}(r1)=d3\}$$

- ▶ Forward: 7 applicable actions
  - five load actions, two move actions
- ▶ Backward:  $g = \{\text{loc}(r1)=d3\}$ 
  - two relevant actions:  
 $\text{move}(r1,d1,d3), \text{move}(r1,d2,d3)$



# Relevance

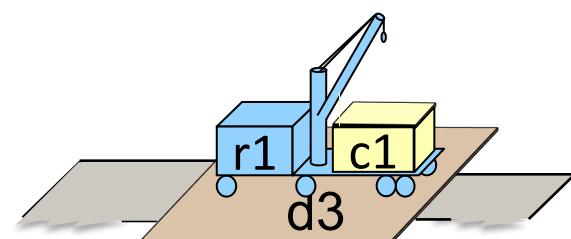
- Idea: when can  $a$  be useful as the last action of a plan to achieve  $g$ ?
  - ▶  $a$  makes at least one atom in  $g$  true that wasn't true already
  - ▶  $a$  doesn't make any part of  $g$  false
- $a$  is *relevant* for  $g = \{x_1=c_1, x_2=c_2, \dots, x_k=c_k\}$  if
  - ▶ at least one atom in  $g$  is also in  $\text{eff}(a)$ 
    - e.g., if  $\text{eff}(a)$  contains  $x_1 \leftarrow c_1$
  - ▶  $\text{eff}(a)$  doesn't make any atom in  $g$  false
    - e.g.,  $\text{eff}(a)$  must not contain  $x_2 \leftarrow c_2'$  (where  $c_2' \neq c_2$ )
  - ▶ whenever  $\text{pre}(a)$  requires an atom of  $g$  to be false,  $\text{eff}(a)$  makes the atom true
    - e.g., if  $\text{pre}(a)$  contains  $x_3 = c_3'$  (where  $c_3' \neq c_3$ ), then  $\text{eff}(a)$  must contain  $x_3 \leftarrow c_3$



$s = \{\text{loc}(c1)=d1, \text{loc}(c2)=d1, \text{loc}(c3)=d1, \text{loc}(r1)=d2, \text{cargo}(r1)=\text{nil}, \text{loc}(r2)=d2, \text{cargo}(r2)=\text{nil}\}$

$\text{load}(r, c, l)$

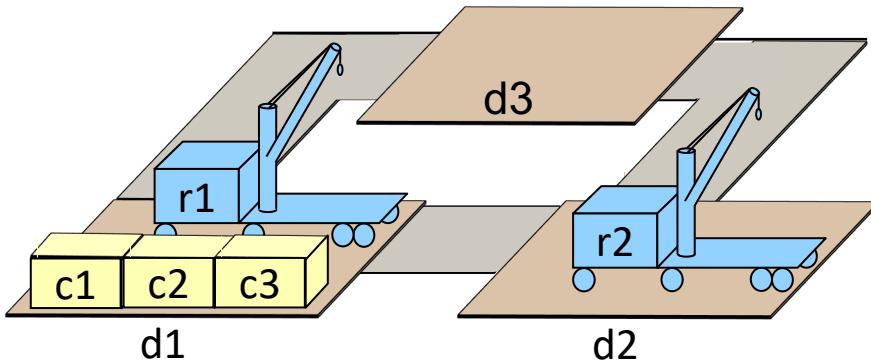
pre:  $\text{cargo}(r)=\text{nil}, \text{loc}(r)=l, \text{loc}(c)=l$   
eff:  $\text{cargo}(r) \leftarrow c, \text{loc}(c) \leftarrow r$



$g = \{\text{cargo}(r1)=c1, \text{loc}(r1)=d3\}$

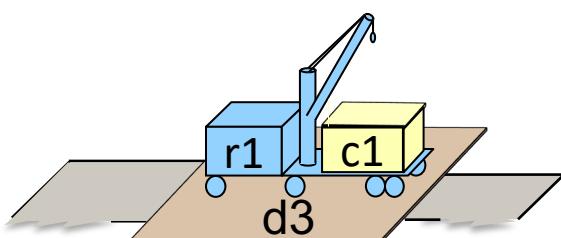


# Relevance



$s = \{\text{loc}(c1)=d1, \text{loc}(c2)=d1, \text{loc}(c3)=d1,$   
 $\text{loc}(r1)=d2, \text{cargo}(r1)=\text{nil},$   
 $\text{loc}(r2)=d2, \text{cargo}(r2)=\text{nil}\}$

$\text{adjacent} = \{(d1,d2), (d1,d3), (d2,d1),$   
 $(d2,d3), (d3,d1), (d3,d2)\}$



$g = \{\text{cargo}(r1)=c1, \text{loc}(r1)=d3\}$

$\text{move}(r,l,m)$

pre:  $\text{loc}(r)=l, \text{adjacent}(l,m)$   
eff:  $\text{loc}(r) \leftarrow m$

$\text{load}(r,c,l)$

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

$\text{put}(r,l,c)$

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

$\text{Range}(r) = \text{Robots} = \{r1,r2\}$

$\text{Range}(l) = \text{Range}(m) = \text{Locs} = \{d1,d2,d3\}$

$\text{Range}(c) = \text{Containers} = \{c1,c2,c3\}$

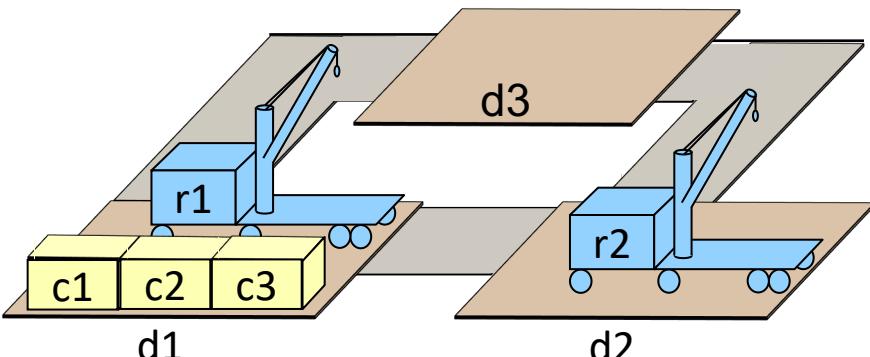
**Poll:** for each action below, is it relevant for  $g$ ?

$\text{load}(r1,c1,d1) \quad \text{load}(r1,c1,d2) \quad \text{put}(r2,c1,d3)$   
 $\text{move}(r1,d1,d3) \quad \text{move}(r1,d3,d1) \quad \text{move}(r1,d2,d3)$



# Inverse State Transitions

- If  $a$  is relevant for  $g$ , then  $\gamma^{-1}(g,a) = \text{pre}(a) \cup (g - \text{eff}(a))$
- If  $a$  isn't relevant for  $g$ , then  $\gamma^{-1}(g,a)$  is undefined
- Example:
  - ▶  $g = \{\text{loc}(c1)=r1\}$
  - ▶ What is  $\gamma^{-1}(g, \text{load}(r1,c1,d3))$ ?
  - ▶ What is  $\gamma^{-1}(g, \text{load}(r2,c1,d1))$ ?



$\text{move}(r,l,m)$

pre:  $\text{loc}(r)=l$ ,  $\text{adjacent}(l,m)$

eff:  $\text{loc}(r) \leftarrow m$

$\text{load}(r,c,l)$

pre:  $\text{cargo}(r)=\text{nil}$ ,  $\text{loc}(r)=l$ ,  $\text{loc}(c)=l$

eff:  $\text{cargo}(r) \leftarrow c$ ,  $\text{loc}(c) \leftarrow r$

$\text{put}(r,l,c)$

pre:  $\text{loc}(r)=l$ ,  $\text{loc}(c)=r$

eff:  $\text{cargo}(r) \leftarrow \text{nil}$ ,  $\text{loc}(c) \leftarrow l$

$\text{Range}(r) = \text{Robots}$

$\text{Range}(l) = \text{Range}(m) = \text{Locs}$

$\text{Range}(c) = \text{Containers}$



# Backward Search

Backward-search( $\Sigma, s_0, g_0$ )

$\pi \leftarrow \langle \rangle; g \leftarrow g_0$

(i)

loop

if  $s_0$  satisfies  $g$  then return  $\pi$

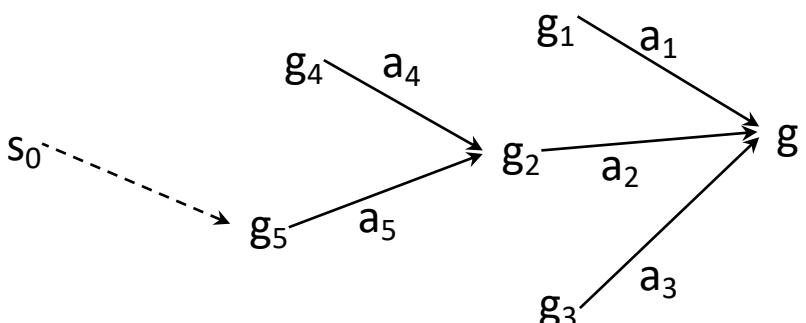
$A' \leftarrow \{a \in A \mid a \text{ is relevant for } g\}$

if  $A' = \emptyset$  then return failure

nondeterministically choose  $a \in A'$

$g \leftarrow \gamma^{-1}(g, a)$  (ii)

$\pi \leftarrow a.\pi$  (iii)



Cycle checking:

- After line (i), put  $Visited \leftarrow \{g_0\}$
- After line (iii), put this:

if  $g \in Visited$  then  
return failure

$Visited \leftarrow Visited \cup \{g\}$

or this:

if  $\exists g' \in Visited \text{ s.t. } g \Rightarrow g'$  then  
return failure

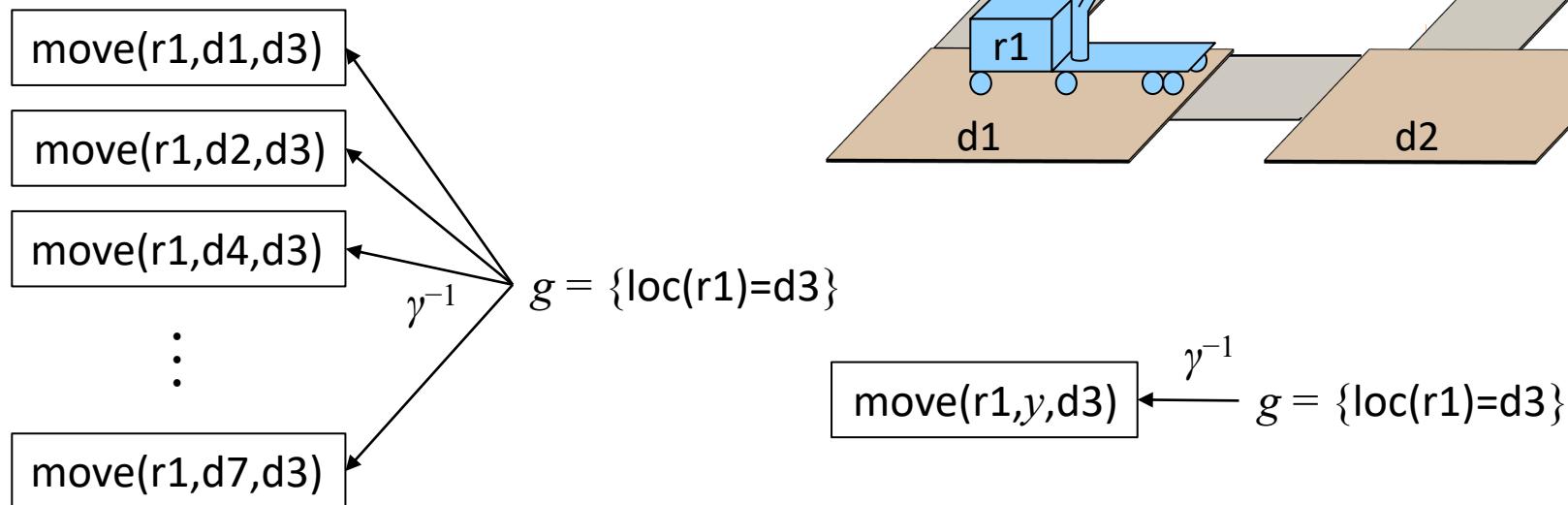
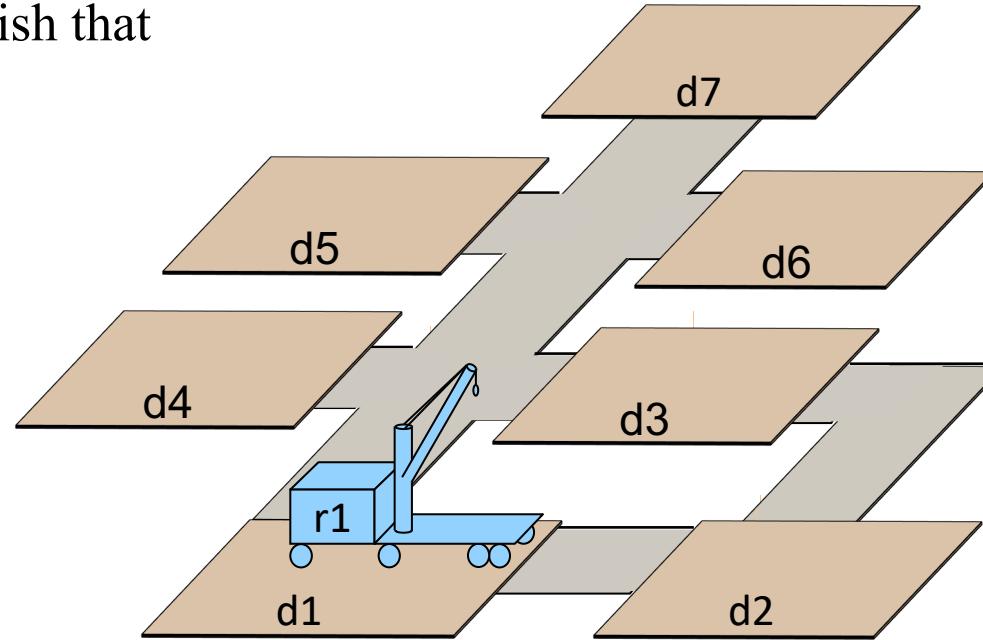
$Visited \leftarrow Visited \cup \{g\}$

- With cycle checking, sound and complete
  - ▶ If  $(\Sigma, s_0, g_0)$  is solvable, then at least one execution trace will find a solution



# Branching Factor

- Motivation for Backward-search was to reduce the branching factor
  - As written, doesn't accomplish that
- Solve this by *lifting*:
  - When possible, leave variables uninstantiated
  - Most implementations of Backward-search do this





# Lifted Backward Search

- Like Backward-search but much smaller branching factor
- Must keep track of what values were substituted for which parameters
  - ▶ I won't discuss the details
  - ▶ PSP (later) does something similar

For classical planning,  
this can be simplified

Backward-search( $\Sigma, s_0, g_0$ )

$\pi \leftarrow \langle \rangle; g \leftarrow g_0$

loop

if  $s_0$  satisfies  $g$  then return  $\pi$

$A' \leftarrow \{a \in A \mid a \text{ is relevant for } g\}$

if  $A' = \emptyset$  then return failure

nondeterministically choose  $a \in A'$

$g \leftarrow \gamma^{-1}(g, a)$

$\pi \leftarrow a.\pi$

Lifted-backward-search( $\mathcal{A}, s_0, g$ )

$\pi \leftarrow$  the empty plan

loop

if  $s_0$  satisfies  $g$  then return  $\pi$

$A \leftarrow \{(o, \theta) \mid o \text{ is a standardization of an action template in } \mathcal{A},$   
 $\theta \text{ is an mgu for an atom of } g \text{ and an atom of effects}^+(o),$   
 $\text{and } \gamma^{-1}(\theta(g), \theta(o)) \text{ is defined}\}$

if  $A = \emptyset$  then return failure

nondeterministically choose a pair  $(o, \theta) \in A$

$\pi \leftarrow$  the concatenation of  $\theta(o)$  and  $\theta(\pi)$

$g \leftarrow \gamma^{-1}(\theta(g), \theta(o))$

# Summary

- 2.4 Backward State-Space Search
  - ▶ Relevance,  $\gamma^{-1}$
  - ▶ Backward search, cycle checking
  - ▶ Lifted backward search (briefly)

# Outline

Chapter 2, part *a* (chap2a.pdf):

- 2.1 State-variable representation
    - Comparison with PDDL
  - 2.2 Forward state-space search
  - 2.6 Incorporating planning into an actor
- 

Chapter 2, part *b* (chap2b.pdf):

- 2.3 Heuristic functions
  - 2.7.7 HTN planning
- 

Chapter 2, part *c* (chap2c.pdf):

- 2.4 Backward search

*Next* → 2.5 Plan-space search

---

Additional slides:

- 2.7.8 LTL\_planning.pdf

## 2.5 Plan-Space Search

- Formulate planning as a constraint satisfaction problem
  - ▶ Use constraint-satisfaction techniques to get solutions that are more flexible than ordinary plans
    - E.g., plans in which the actions are partially ordered
    - Postpone ordering decisions until the plan is being executed
      - ▶ the actor may have a better idea about which ordering is best
- First step toward temporal planning (Chapter 4)
- Basic idea:
  - ▶ Backward search from the goal
  - ▶ Each node of the search space is a *partial plan* that contains *flaws*
    - Remove the flaws by making *refinements*
  - ▶ If successful, we'll get a *partially ordered* solution

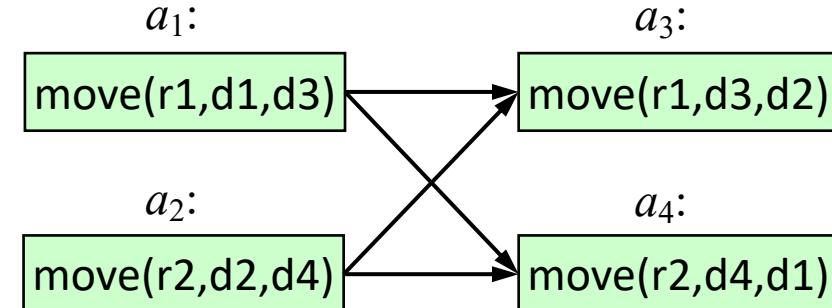
# Definitions

- *Partially ordered plan*
  - ▶ partially ordered set of nodes
  - ▶ each node contains an action
- *Partially ordered solution* for a planning problem  $P$ 
  - ▶ partially ordered plan  $\pi$  such that every total ordering of  $\pi$  is a solution for  $P$

**Qn.** Let  $P$  be the planning problem at right, and  $\pi$  be the partially ordered plan below. Is  $\pi$  a partially ordered solution for  $P$ ?

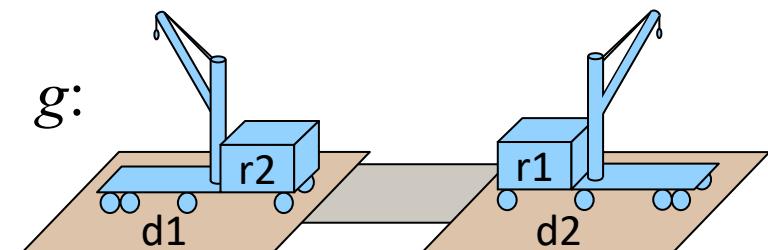
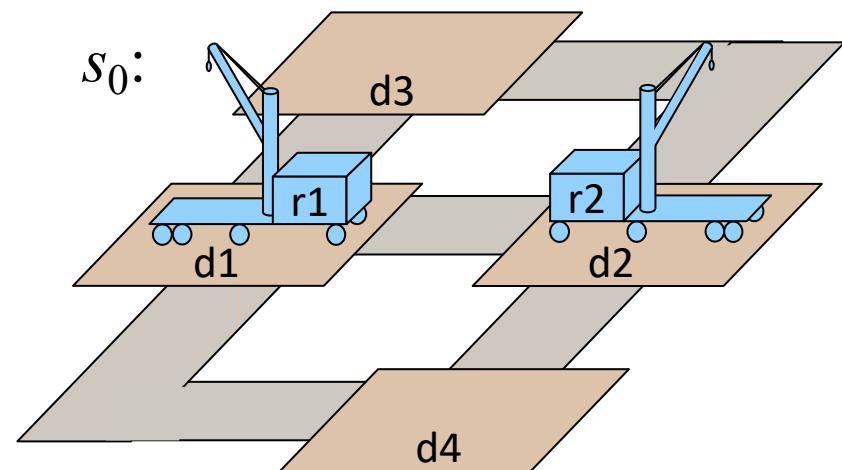
precedence  
constraints:

$$a_1 \prec a_3, a_1 \prec a_4, \\ a_2 \prec a_3, a_2 \prec a_4$$



$\text{move}(r, d, d')$   
pre:  $\text{loc}(r) = d$ ,  $\text{occupied}(d') = \text{nil}$   
eff:  $\text{loc}(r) \leftarrow d'$ ,  $\text{occupied}(d') \leftarrow r$ ,  $\text{occupied}(d) \leftarrow \text{nil}$

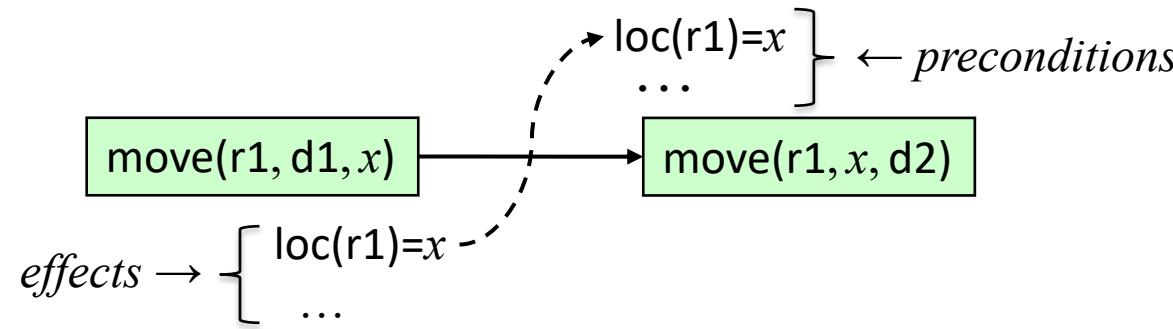
$\text{Range}(r) = \text{Robots}$   
 $\text{Range}(d) = \text{Range}(d') = \text{Docks}$



# Definitions

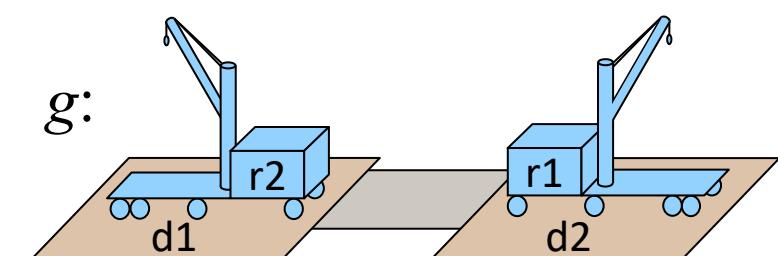
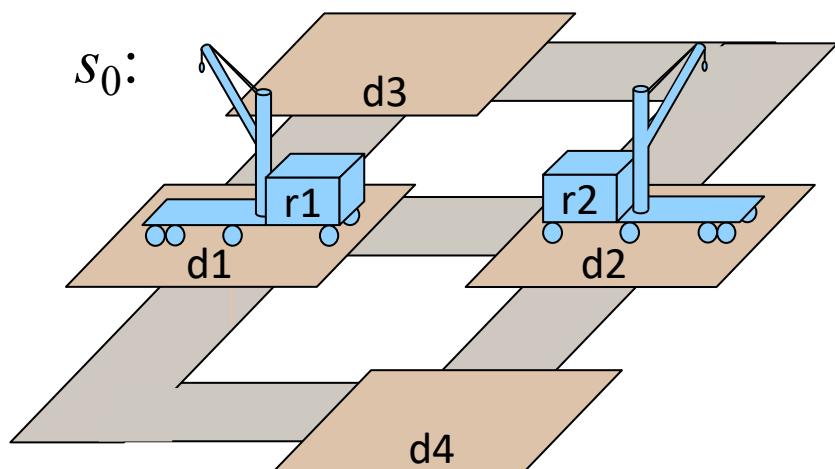
- *Partial plan*

- ▶ partially ordered set of nodes that contain *partially instantiated actions*
- ▶ *inequality constraints*, e.g.  $z \neq x$  or  $w \neq p_1$
- ▶ *causal links* (dashed arcs)
  - constraint: action  $a$  *must* be the action that establishes action  $b$ 's precondition  $p$



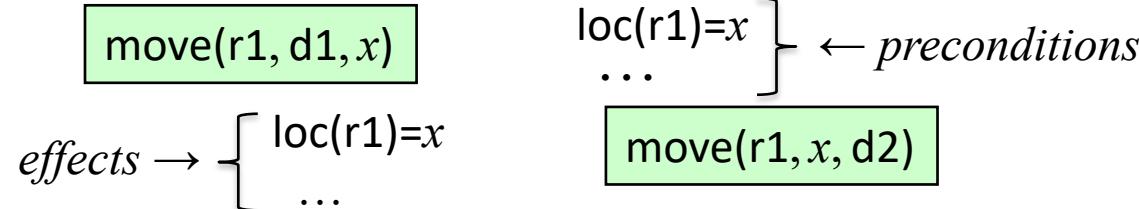
$\text{move}(r, d, d')$   
 pre:  $\text{loc}(r) = d$ ,  $\text{occupied}(d') = \text{nil}$   
 eff:  $\text{loc}(r) \leftarrow d'$ ,  $\text{occupied}(d') \leftarrow r$ ,  $\text{occupied}(d) \leftarrow \text{nil}$

$\text{Range}(r) = \text{Robots}$   
 $\text{Range}(d) = \text{Range}(d') = \text{Docks}$



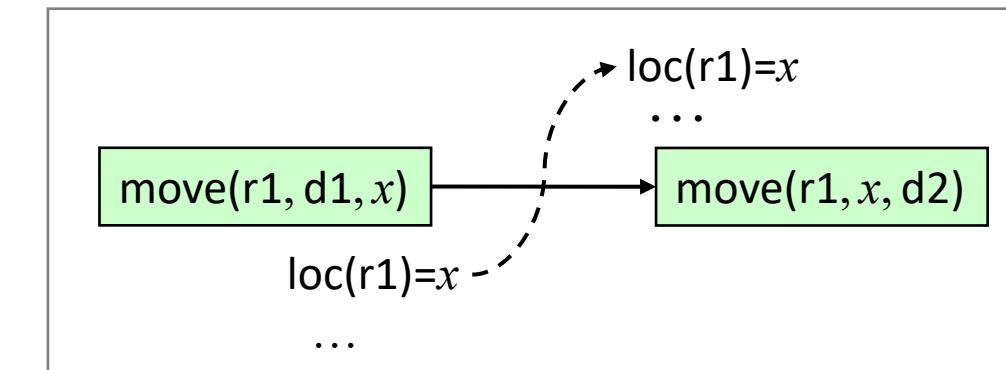
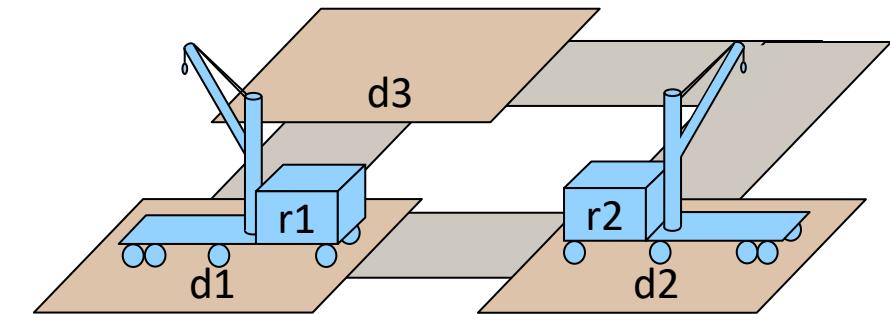
# Flaws: 1. Open Goals

- Action  $b$ , precondition  $p$ 
  - ▶  $p$  is an *open goal* if there is no causal link for  $p$
- Resolve the flaw by creating a causal link
  - ▶ Find an action  $a$  (either already in  $\pi$ , or add it to  $\pi$ ) that can *establish*  $p$ 
    - can precede  $b$
    - can have  $p$  as an effect
  - ▶ Do substitutions on variables to make  $a$  assert  $p$
  - ▶ Add an ordering constraint  $a \prec b$
  - ▶ Create a causal link from  $a$  to  $p$



`move( $r$ ,  $d$ ,  $d'$ )`  
 pre:  $\text{loc}(r) = d$ ,  $\text{occupied}(d') = \text{nil}$   
 eff:  $\text{loc}(r) \leftarrow d'$ ,  $\text{occupied}(d') \leftarrow r$ ,  $\text{occupied}(d) \leftarrow \text{nil}$

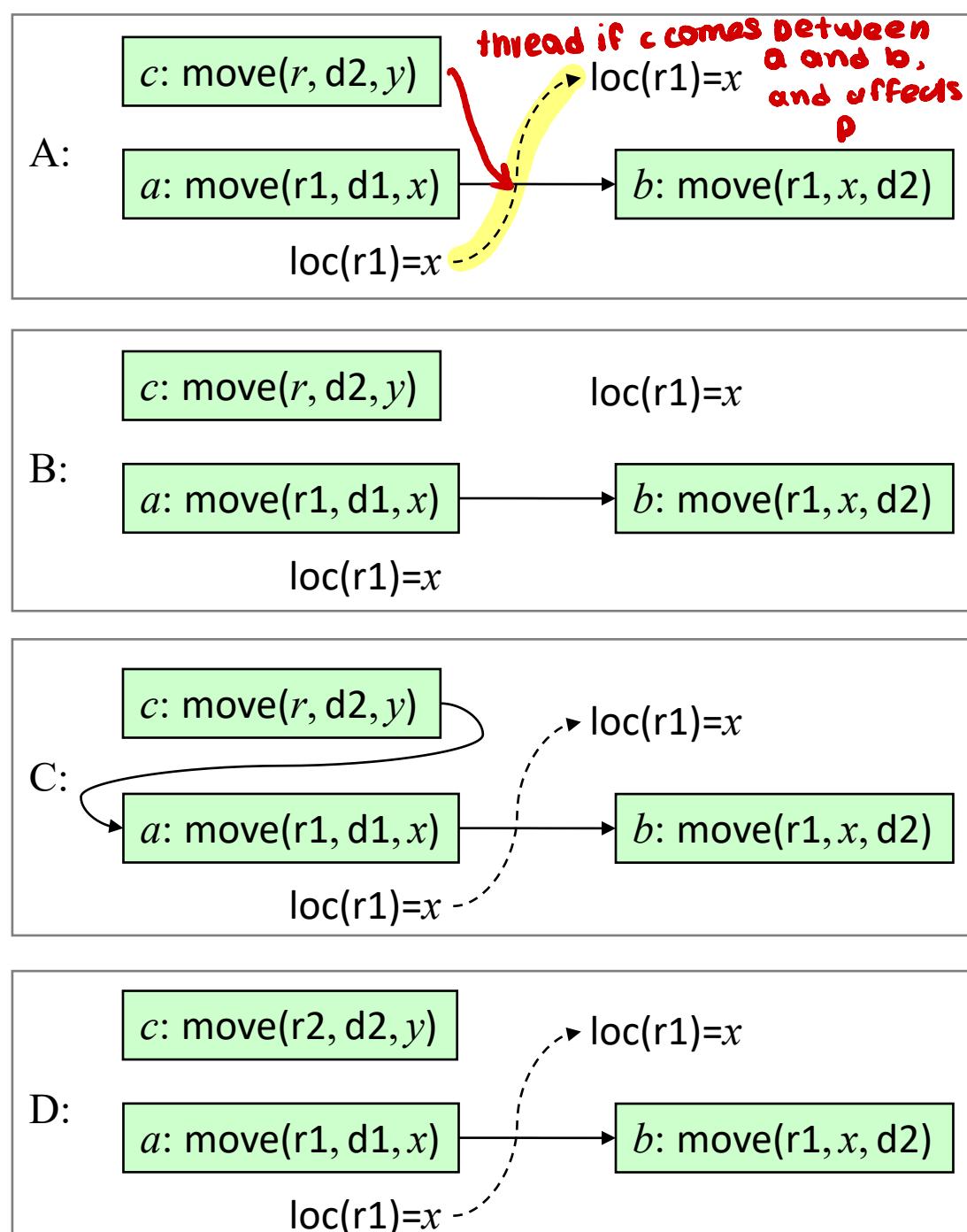
Range( $r$ ) = Robots  
 Range( $d$ ) = Range( $d'$ ) = Docks



## Flaws: 2. Threats

- Let  $l$  be a causal link from an effect of action  $a$  to a precondition  $p$  of action  $b$
- Action  $c$  *threatens*  $l$  if  $c$  may come between  $a$  and  $b$  and  $c$  may affect  $p$ 
  - “ $c$  may come between  $a$  and  $b$ ” means the plan’s current ordering constraints don’t prevent it
    - plan doesn’t already have  $c \prec a$  or  $b \prec c$
  - “ $c$  may affect  $p$ ” means
    - can substitute values for variables such that  $c$ ’s effects either make  $p$  true or make  $p$  false

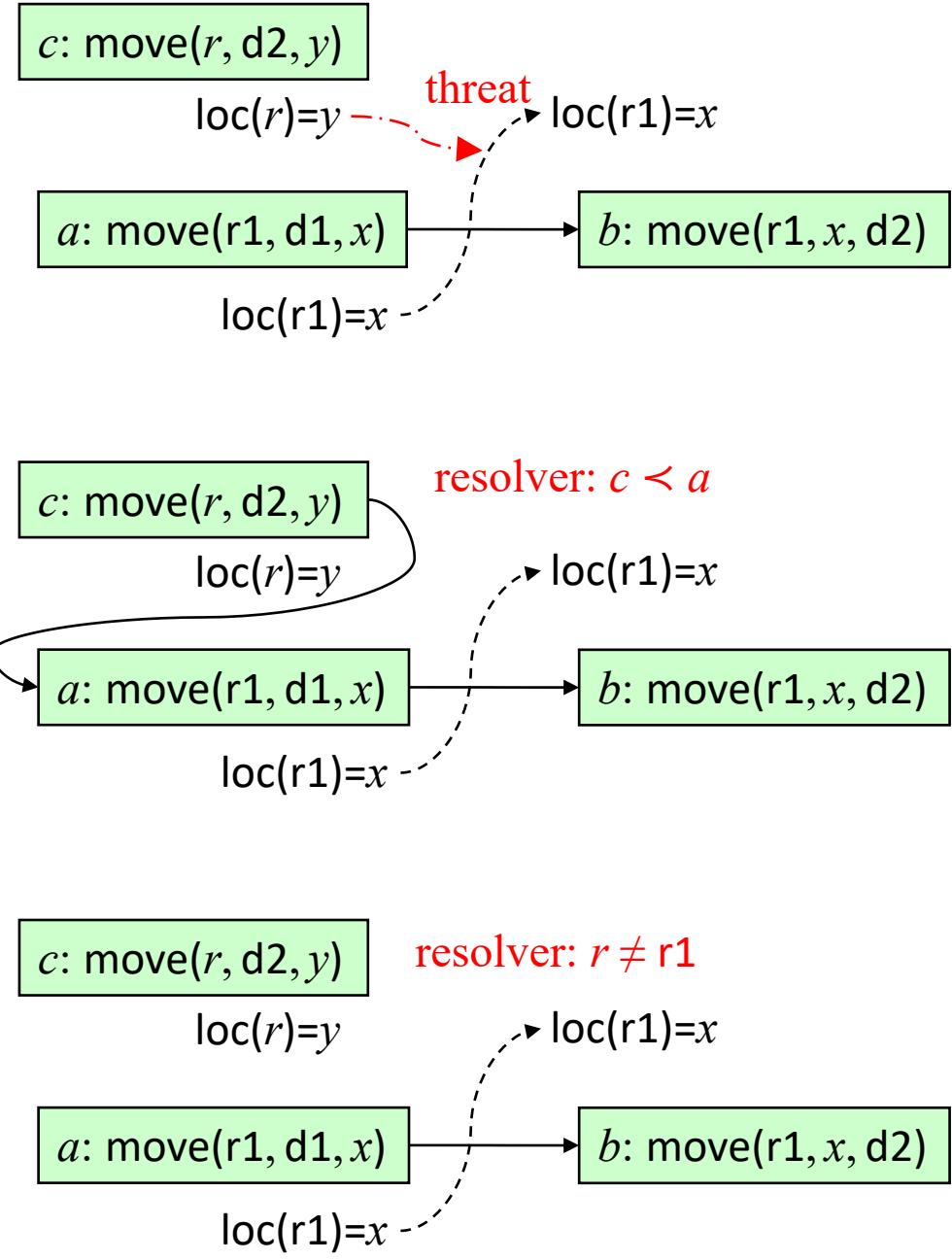
**Poll.** In each of the cases at right, does action  $c$  threaten the causal link?



# Resolving Threats

- Suppose action  $c$  threatens a causal link  $l$  from an effect of action  $a$  to a precondition  $p$  of action  $b$
- Three possible resolvers:
  - Add a precedence constraint  $c \prec a$
  - Add a precedence constraint  $b \prec c$
  - Add inequality constraints that prevent  $c$  from affecting  $p$
- Each of these is applicable iff it doesn't make the plan inconsistent
  - e.g., 2 isn't applicable if the plan already has  $c \prec b$

inconsistency



$\text{PSP}(\Sigma, \pi)$

loop

- if  $\text{Flaws}(\pi) = \emptyset$  then return  $\pi$
- arbitrarily select  $f \in \text{Flaws}(\pi)$
- $R \leftarrow \{\text{all feasible resolvers for } f\}$
- if  $R = \emptyset$  then return failure
- nondeterministically choose  $\rho \in R$
- $\pi \leftarrow \rho(\pi)$

return  $\pi$

- 2 open goals
- no threats

$a_0$

$\text{loc}(r1) = d1$

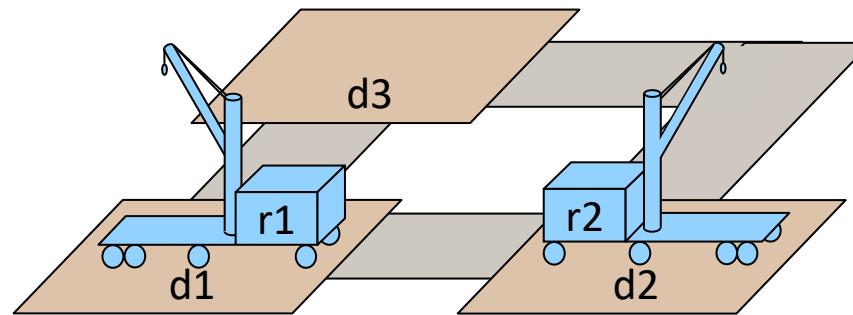
$\text{loc}(r2) = d2$

$\text{occupied}(d3) = \text{nil}$

$\text{occupied}(d1) = r1$

$\text{occupied}(d2) = r2$

## PSP Algorithm



*select* →  $\text{loc}(r1) = d2$   
 $\text{loc}(r2) = d1$

$a_g$

$\text{move}(r, d, d')$

pre:  $\text{loc}(r) = d$ ,  $\text{occupied}(d') = \text{nil}$

eff:  $\text{loc}(r) \leftarrow d'$ ,  $\text{occupied}(d') = r$ ,  $\text{occupied}(d) = \text{nil}$

$\text{PSP}(\Sigma, \pi)$

loop

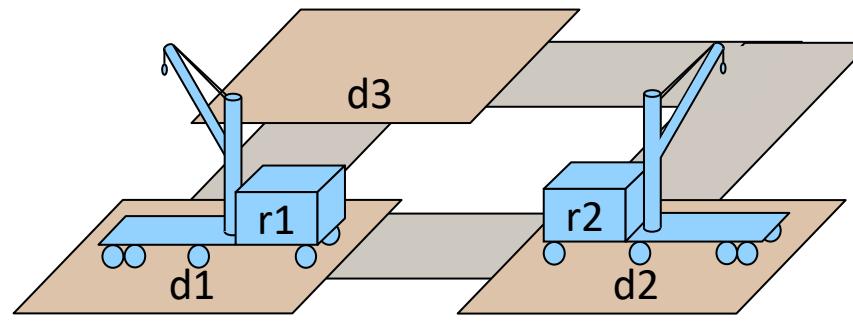
```

if  $\text{Flaws}(\pi) = \emptyset$  then return  $\pi$ 
arbitrarily select  $f \in \text{Flaws}(\pi)$ 
 $R \leftarrow \{\text{all feasible resolvers for } f\}$ 
if  $R = \emptyset$  then return failure
nondeterministically choose  $\rho \in R$ 
 $\pi \leftarrow \rho(\pi)$ 
return  $\pi$ 

```

- 3 open goals
- no threats

# PSP Algorithm



$\text{loc}(r1) = d$   
 $\text{occupied}(d2) = \text{nil}$

$a_1 = \text{move}(r1, d, d2)$   
 $\text{loc}(r1) = d2$   
 $\text{occupied}(d) = \text{nil}$   
 $\text{occupied}(d2) = r1$

*only resolver:  
causal link from  
a new action*

$\text{loc}(r1) = d2$   
 $\text{loc}(r2) = d1$   
 $\uparrow$   
 $a_g$

*select*

$a_0$   
 $\text{loc}(r1) = d1$   
 $\text{loc}(r2) = d2$   
 $\text{occupied}(d3) = \text{nil}$   
 $\text{occupied}(d1) = r1$   
 $\text{occupied}(d2) = r2$

*for every action  $a$ ,*  
 $a_0 < a < a_g$

$\text{move}(r, d, d')$

pre:  $\text{loc}(r) = d$ ,  $\text{occupied}(d') = \text{nil}$

eff:  $\text{loc}(r) \leftarrow d'$ ,  $\text{occupied}(d') = r$ ,  $\text{occupied}(d) = \text{nil}$

**Poll.** Above, I said “only resolver”. Is that correct?

$\text{PSP}(\Sigma, \pi)$

loop

```

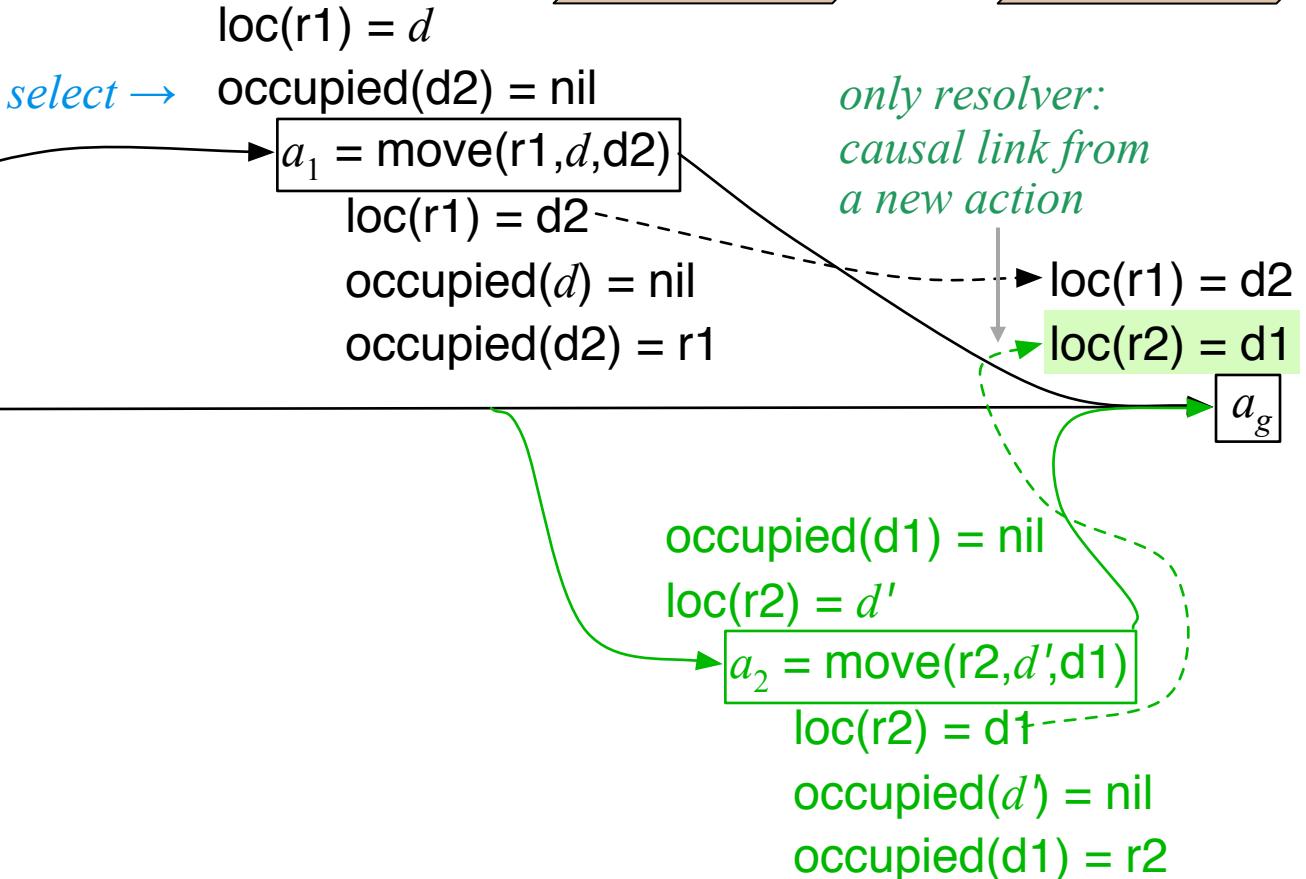
if  $\text{Flaws}(\pi) = \emptyset$  then return  $\pi$ 
arbitrarily select  $f \in \text{Flaws}(\pi)$ 
 $R \leftarrow \{\text{all feasible resolvers for } f\}$ 
if  $R = \emptyset$  then return failure
nondeterministically choose  $\rho \in R$ 
 $\pi \leftarrow \rho(\pi)$ 
return  $\pi$ 

```

- 4 open goals
- no threats

# PSP Algorithm

$a_0$   
 $\text{loc}(r1) = d1$   
 $\text{loc}(r2) = d2$   
 $\text{occupied}(d3) = \text{nil}$   
 $\text{occupied}(d1) = r1$   
 $\text{occupied}(d2) = r2$



$\text{move}(r, d, d')$

pre:  $\text{loc}(r) = d$ ,  $\text{occupied}(d') = \text{nil}$

eff:  $\text{loc}(r) \leftarrow d'$ ,  $\text{occupied}(d') = r$ ,  $\text{occupied}(d) = \text{nil}$

$\text{PSP}(\Sigma, \pi)$

loop

```

if  $\text{Flaws}(\pi) = \emptyset$  then return  $\pi$ 
arbitrarily select  $f \in \text{Flaws}(\pi)$ 
 $R \leftarrow \{\text{all feasible resolvers for } f\}$ 
if  $R = \emptyset$  then return failure
nondeterministically choose  $\rho \in R$ 
 $\pi \leftarrow \rho(\pi)$ 
return  $\pi$ 

```

- 5 open goals
- 1 threat

## PSP Algorithm

*only resolver:  
causal link from  
a new action*

*select*

$\text{loc}(r1) = d$

$\text{occupied}(d2) = \text{nil}$

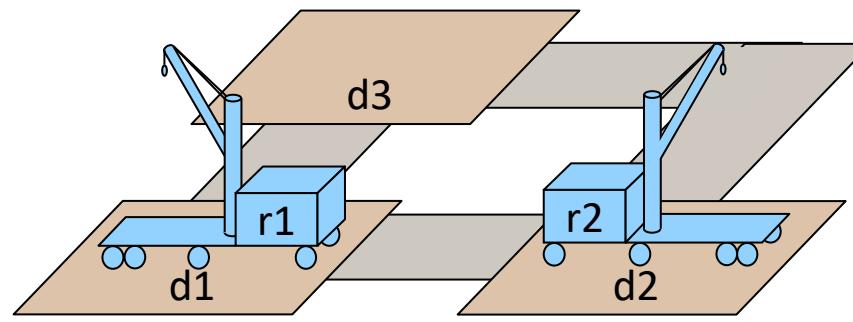
$a_1 = \text{move}(r1, d, d2)$

$\text{loc}(r1) = d2$

$\text{occupied}(d) = \text{nil}$

$\text{occupied}(d2) = r1$

$d3$



$a_0$   
 $\text{loc}(r1) = d1$   
 $\text{loc}(r2) = d2$   
 $\text{occupied}(d3) = \text{nil}$   
 $\text{occupied}(d1) = r1$   
 $\text{occupied}(d2) = r2$

$\text{loc}(r) = d2$

$\text{occupied}(d'') = \text{nil}$

$a_3 = \text{move}(r, d2, d'')$

$\text{loc}(r) = d''$

$\text{occupied}(d2) = \text{nil}$

$\text{occupied}(d'') = r$

$\text{loc}(r1) = d2$

$\text{loc}(r2) = d1$

*threat*

$\text{occupied}(d1) = \text{nil}$

$\text{loc}(r2) = d'$

$a_2 = \text{move}(r2, d', d1)$

$\text{loc}(r2) = d1$

$\text{occupied}(d') = \text{nil}$

$\text{occupied}(d1) = r2$

$\text{move}(r, d, d')$

pre:  $\text{loc}(r) = d$ ,  $\text{occupied}(d') = \text{nil}$

eff:  $\text{loc}(r) \leftarrow d'$ ,  $\text{occupied}(d') = r$ ,  $\text{occupied}(d) = \text{nil}$

**Poll:** does  $a_3$  threaten  $a_1$ 's precondition  $\text{loc}(r1)=d$ ?

$\text{PSP}(\Sigma, \pi)$

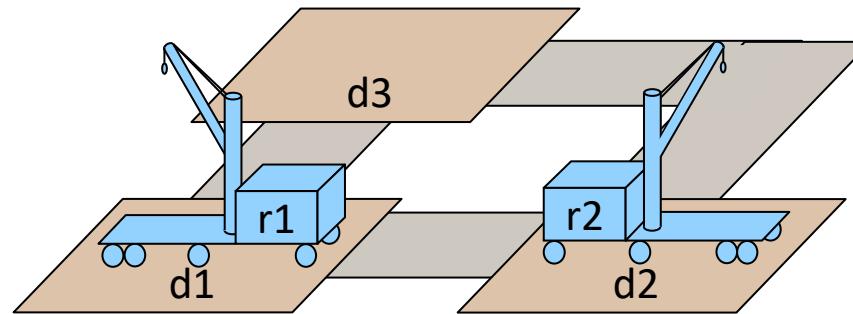
loop

if  $\text{Flaws}(\pi) = \emptyset$  then return  $\pi$   
arbitrarily select  $f \in \text{Flaws}(\pi)$   
 $R \leftarrow \{\text{all feasible resolvers for } f\}$   
if  $R = \emptyset$  then return failure  
nondeterministically choose  $\rho \in R$   
 $\pi \leftarrow \rho(\pi)$

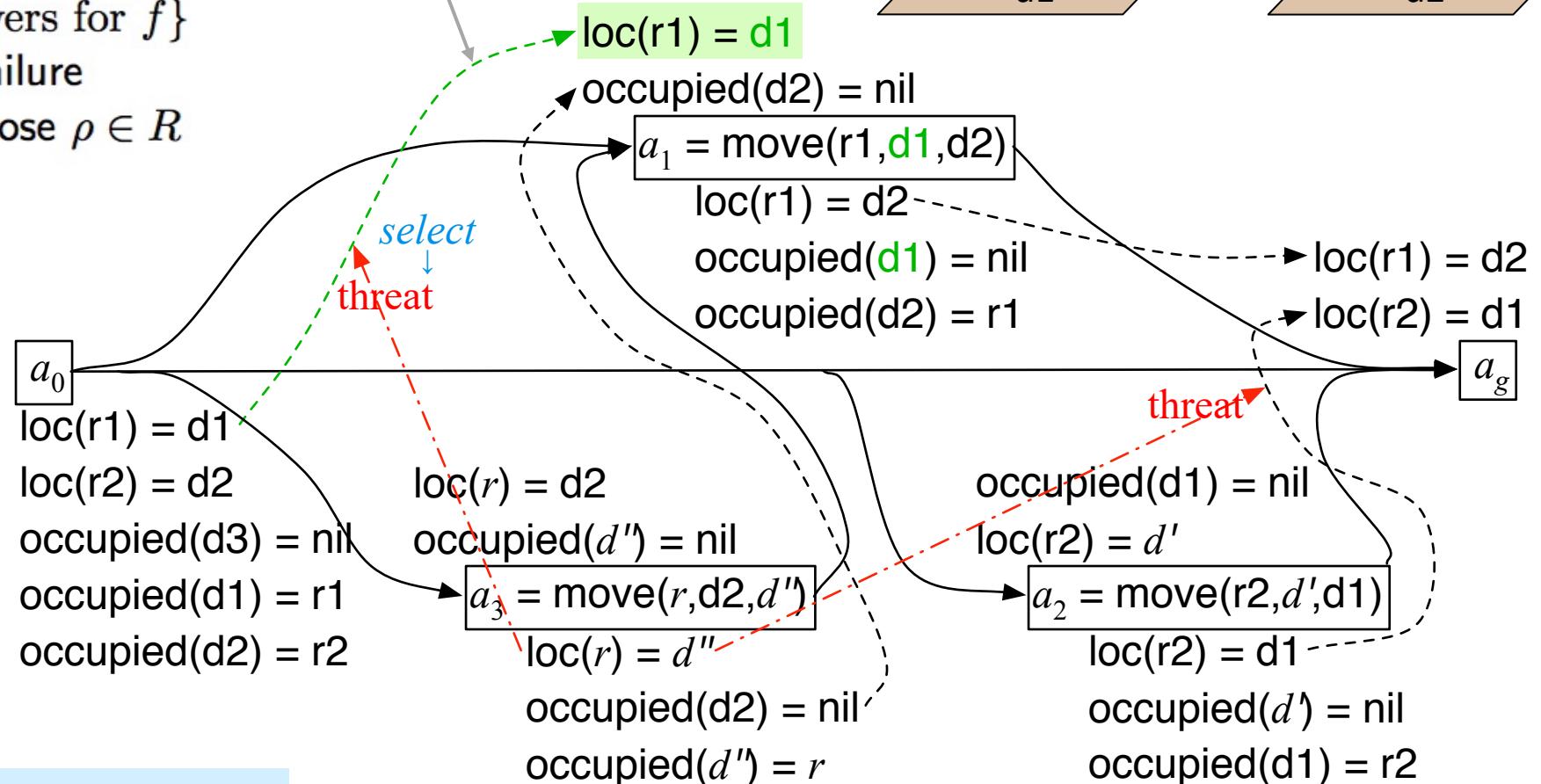
return  $\pi$

- 4 open goals
- 2 threats

# PSP Algorithm



*causal link from  $a_0$ , with substitution  $d \leftarrow d_1$*



$\text{move}(r, d, d')$

pre:  $\text{loc}(r) = d$ ,  $\text{occupied}(d') = \text{nil}$

eff:  $\text{loc}(r) \leftarrow d'$ ,  $\text{occupied}(d') = r$ ,  $\text{occupied}(d) = \text{nil}$

Poll: does  $a_3$  threaten the causal link for  $a_g$ 's precondition  $\text{loc}(r1)=d2$ ?

$\text{PSP}(\Sigma, \pi)$

loop

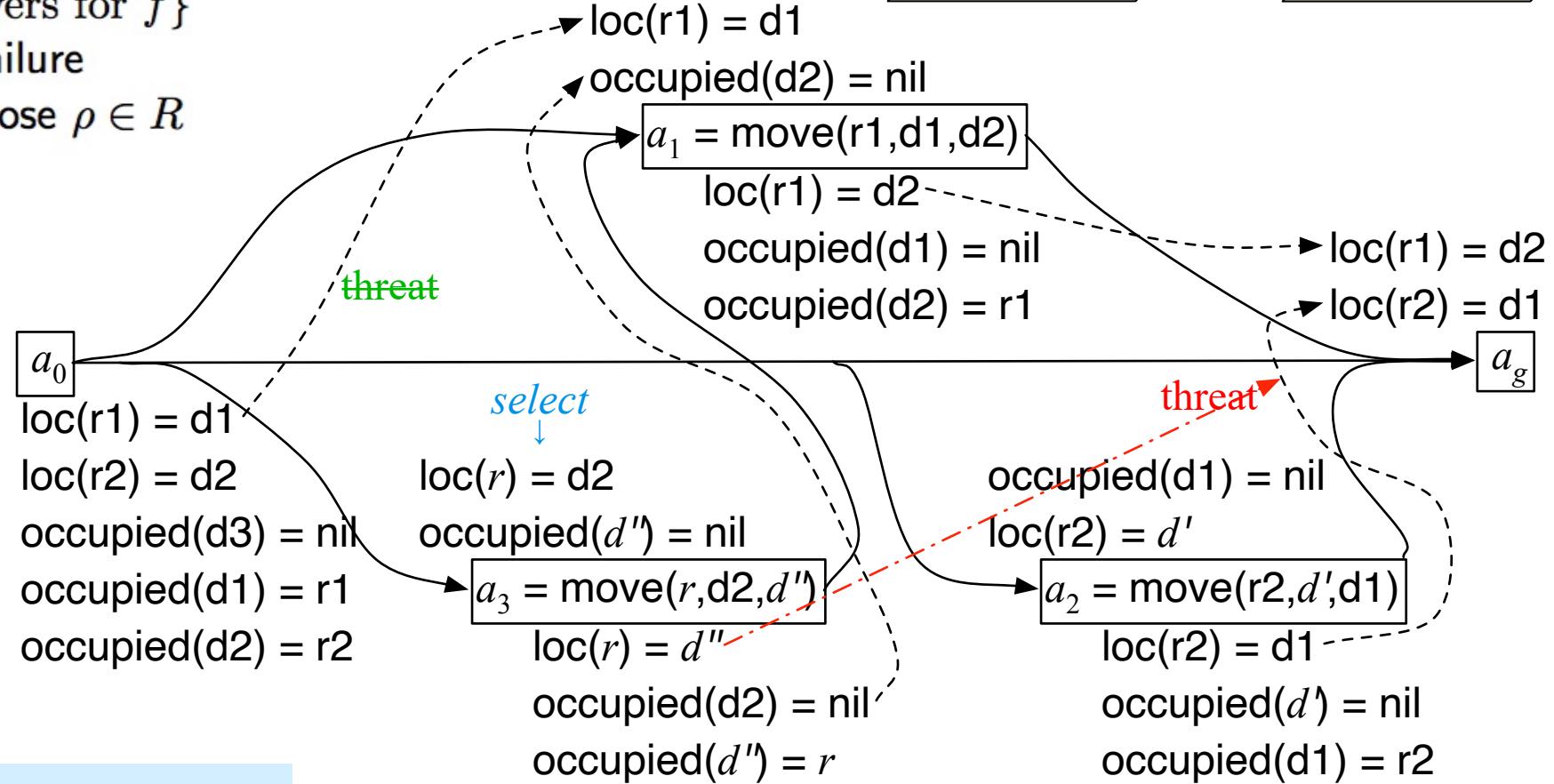
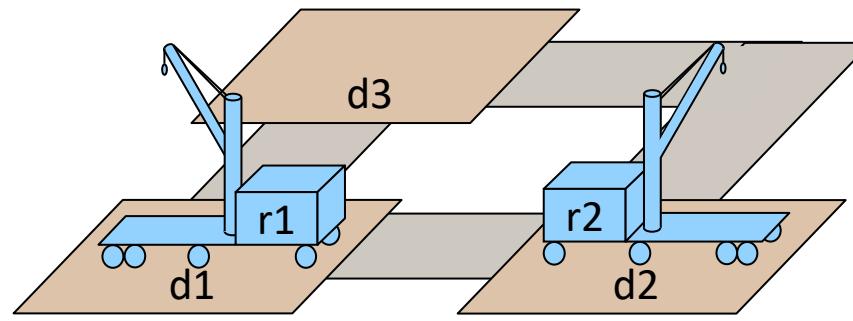
if  $\text{Flaws}(\pi) = \emptyset$  then return  $\pi$   
 arbitrarily select  $f \in \text{Flaws}(\pi)$   
 $R \leftarrow \{\text{all feasible resolvers for } f\}$   
 if  $R = \emptyset$  then return failure  
 nondeterministically choose  $\rho \in R$   
 $\pi \leftarrow \rho(\pi)$

return  $\pi$

- 4 open goals
- 1 threat

Constraint:  $r \neq r1$

# PSP Algorithm



$\text{move}(r, d, d')$

pre:  $\text{loc}(r) = d$ ,  $\text{occupied}(d') = \text{nil}$

eff:  $\text{loc}(r) \leftarrow d'$ ,  $\text{occupied}(d') = r$ ,  $\text{occupied}(d) = \text{nil}$

$\text{PSP}(\Sigma, \pi)$

loop

```

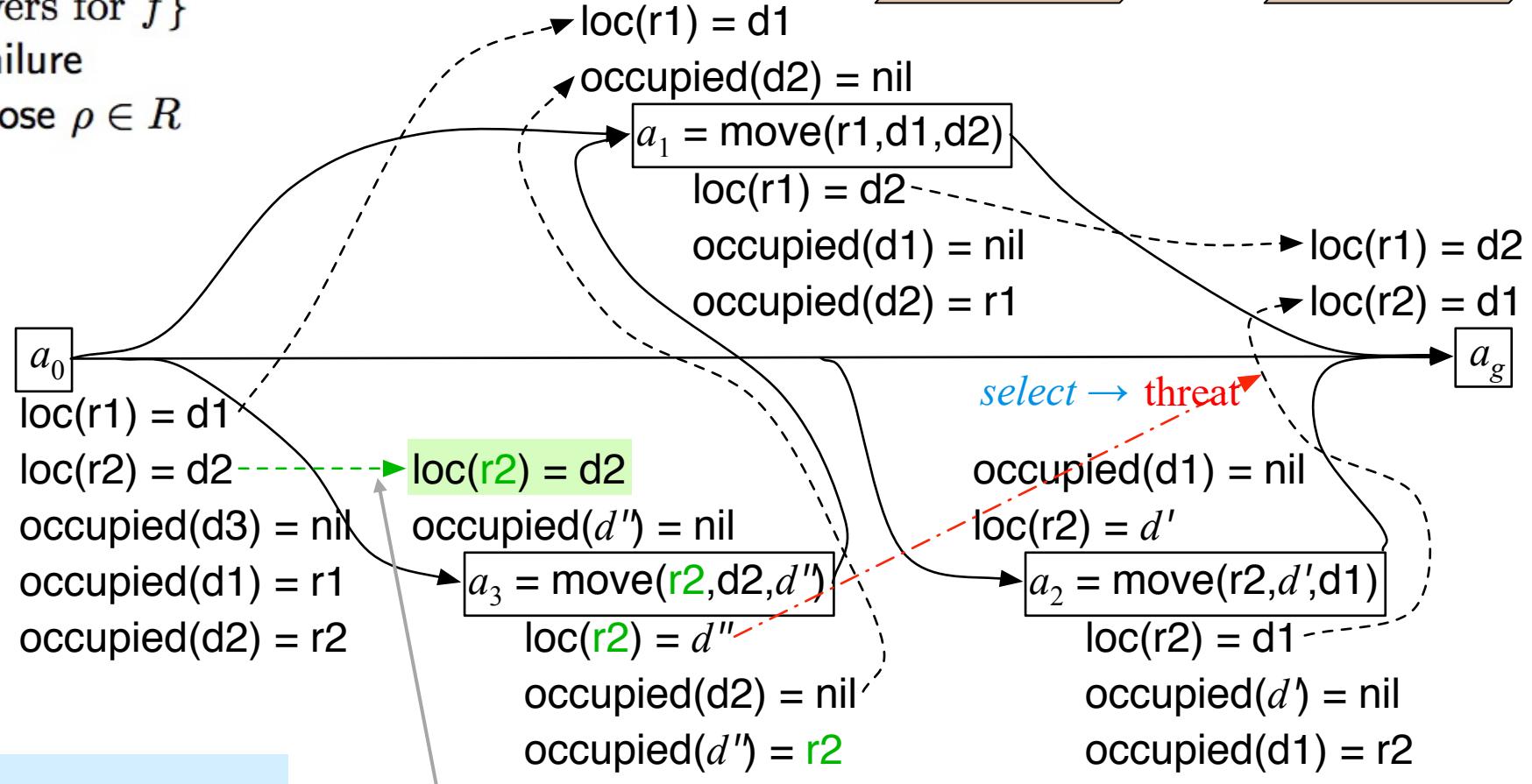
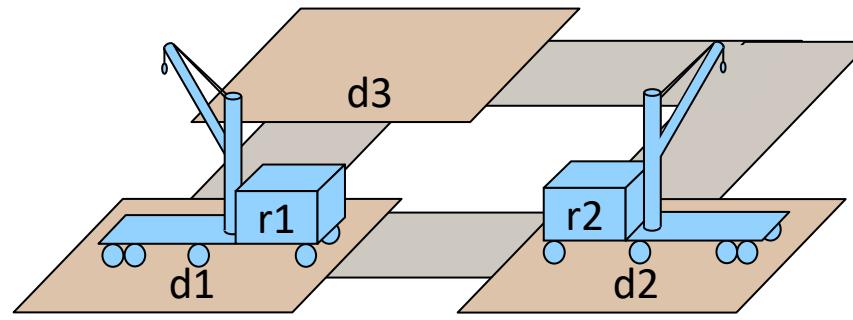
if  $\text{Flaws}(\pi) = \emptyset$  then return  $\pi$ 
arbitrarily select  $f \in \text{Flaws}(\pi)$ 
 $R \leftarrow \{\text{all feasible resolvers for } f\}$ 
if  $R = \emptyset$  then return failure
nondeterministically choose  $\rho \in R$ 
 $\pi \leftarrow \rho(\pi)$ 
return  $\pi$ 

```

- 3 open goals
- 1 threat

Constraint:  $r \neq r_1$

# PSP Algorithm



$\text{move}(r, d, d')$

pre:  $\text{loc}(r) = d$ ,  $\text{occupied}(d') = \text{nil}$

eff:  $\text{loc}(r) \leftarrow d'$ ,  $\text{occupied}(d') = r$ ,  $\text{occupied}(d) = \text{nil}$

causal link from  $a_0$   
with substitution  $r \leftarrow r_2$

$\text{PSP}(\Sigma, \pi)$

loop

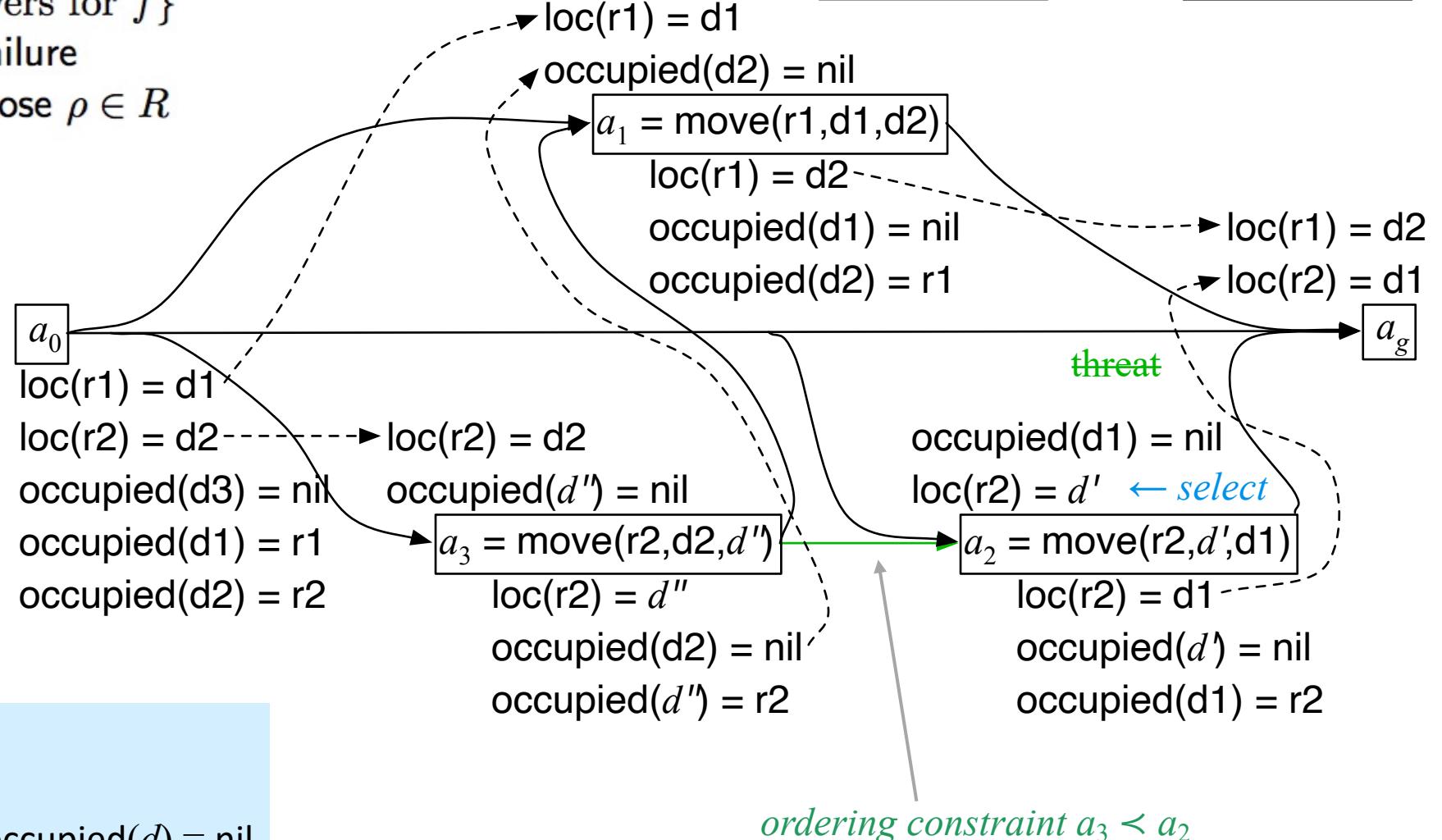
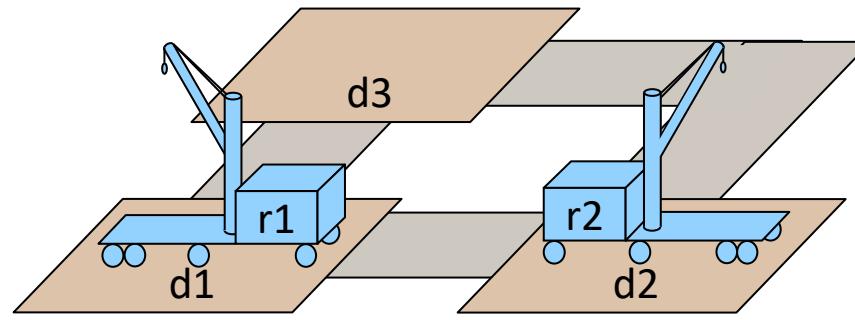
```

if  $\text{Flaws}(\pi) = \emptyset$  then return  $\pi$ 
arbitrarily select  $f \in \text{Flaws}(\pi)$ 
 $R \leftarrow \{\text{all feasible resolvers for } f\}$ 
if  $R = \emptyset$  then return failure
nondeterministically choose  $\rho \in R$ 
 $\pi \leftarrow \rho(\pi)$ 
return  $\pi$ 

```

- 3 open goals
- no threats

# PSP Algorithm



$$\text{PSP}(\Sigma, \pi)$$

loop

```

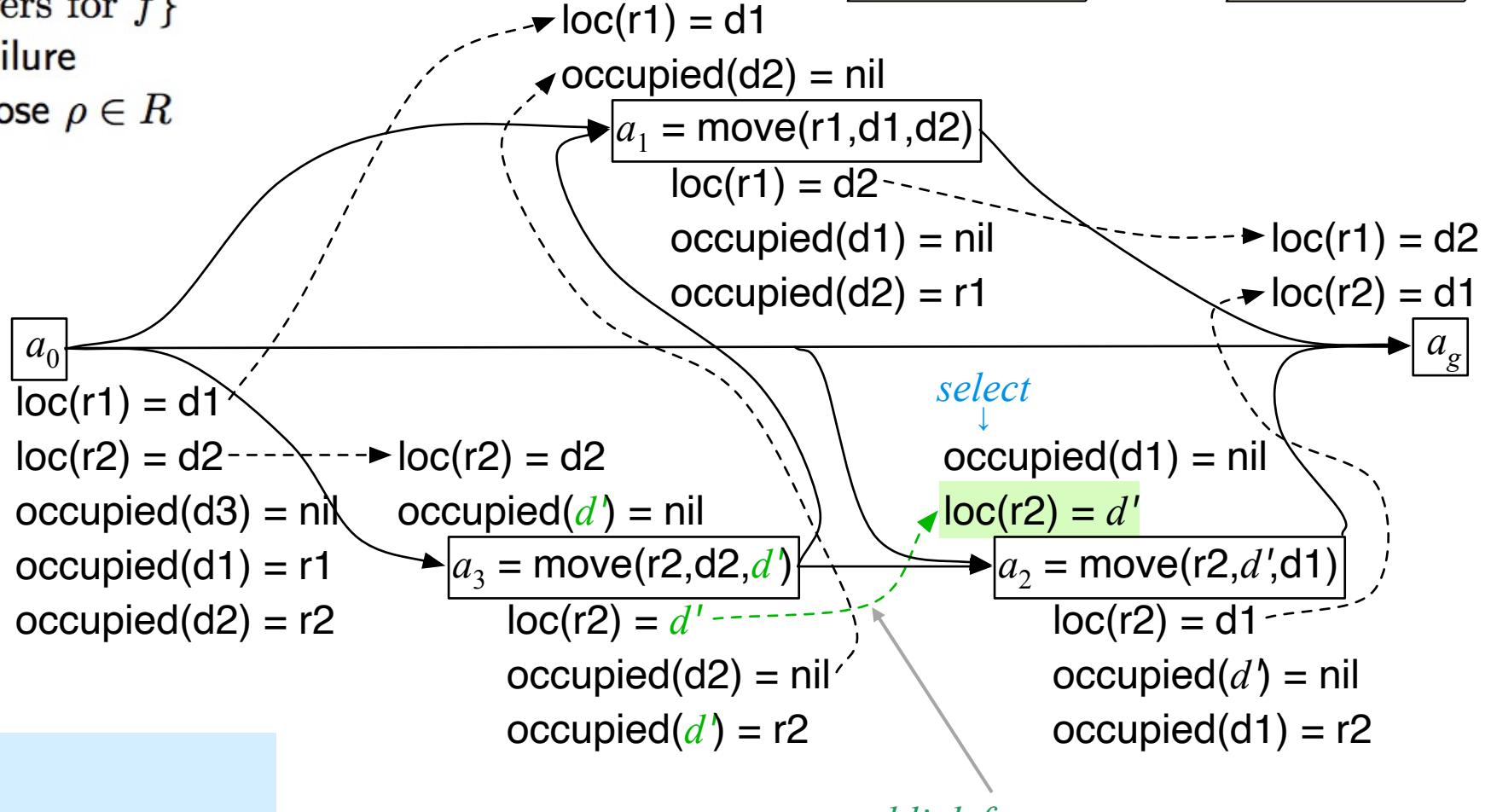
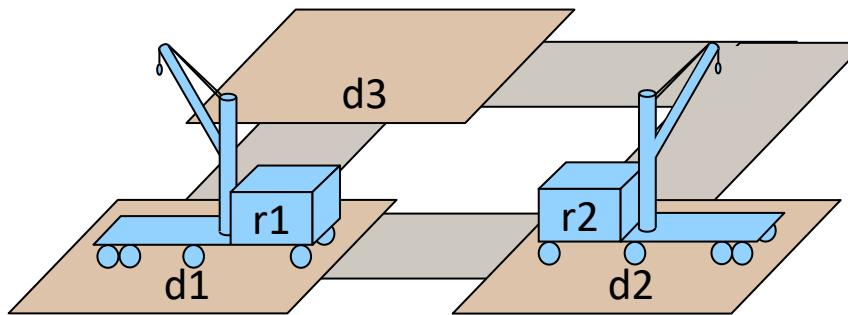
if  $Flaws(\pi) = \emptyset$  then return  $\pi$ 
arbitrarily select  $f \in Flaws(\pi)$ 
 $R \leftarrow \{ \text{all feasible resolvers for } f \}$ 
if  $R = \emptyset$  then return failure
nondeterministically choose  $\rho \in R$ 
 $\pi \leftarrow \rho(\pi)$ 
return  $\pi$ 

```

return  $\pi$

- 2 open goals
  - no threats

# PSP Algorithm



**move( $r$ ,  $d$ ,  $d'$ )**

pre:  $\text{loc}(r) = d$ ,  $\text{occupied}(d') = \text{nil}$

**eff:**  $\text{loc}(r) \leftarrow d'$ ,  $\text{occupied}(d') = r$ ,  $\text{occupied}(d) = \text{nil}$

$$\text{PSP}(\Sigma, \pi)$$

loop

```

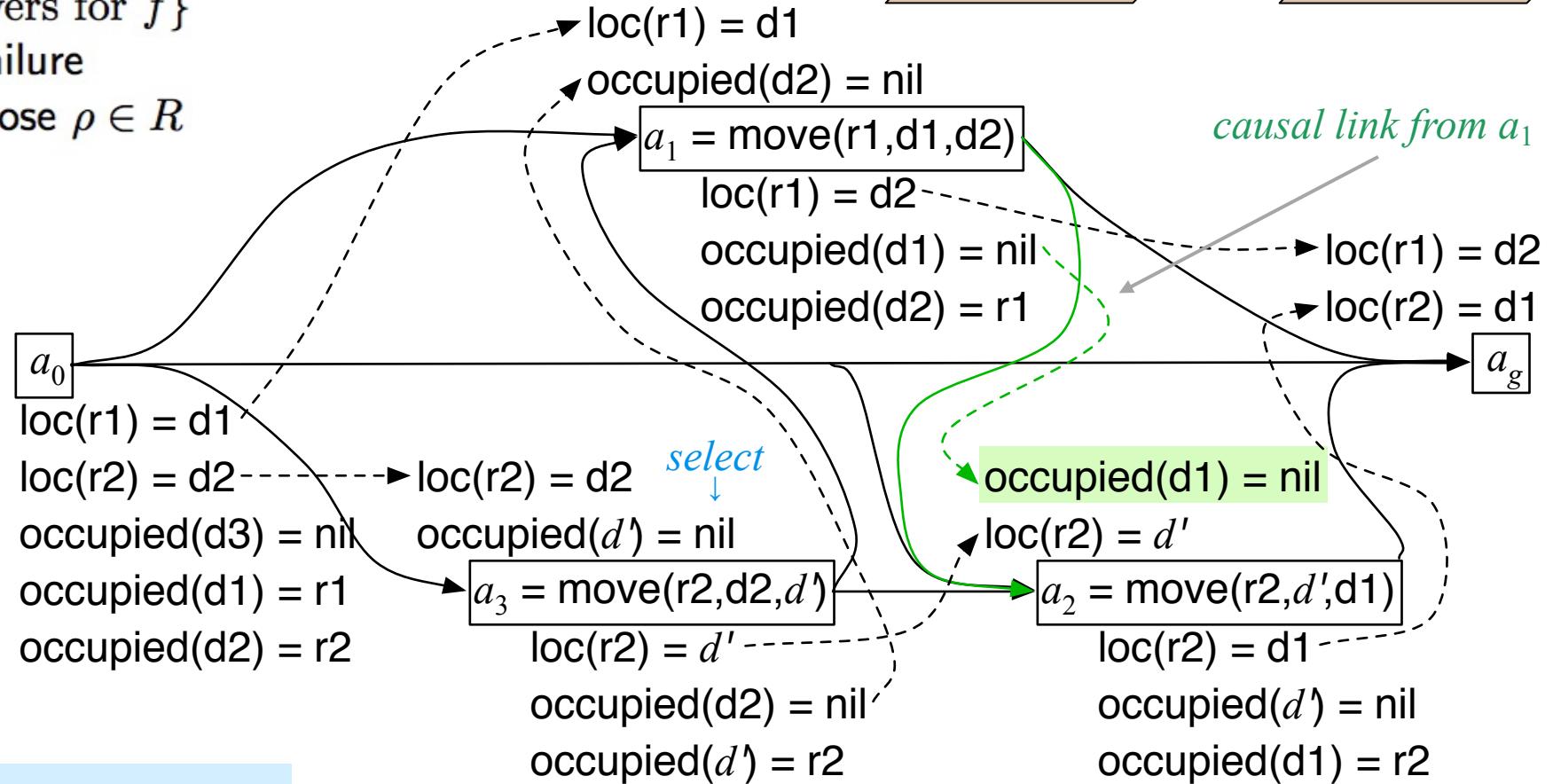
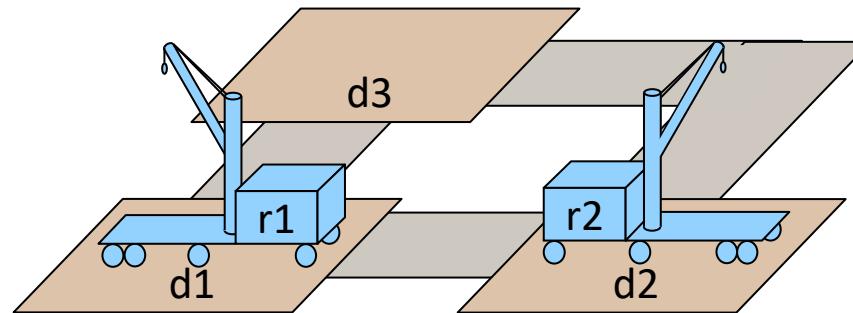
if  $Flaws(\pi) = \emptyset$  then return  $\pi$ 
arbitrarily select  $f \in Flaws(\pi)$ 
 $R \leftarrow \{\text{all feasible resolvers for } f\}$ 
if  $R = \emptyset$  then return failure
nondeterministically choose  $\rho \in R$ 
 $\pi \leftarrow \rho(\pi)$ 

```

return  $\pi$

- 1 open goal
  - no threats

# PSP Algorithm



**move**( $r$ ,  $d$ ,  $d'$ )

pre:  $\text{loc}(r) = d$ ,  $\text{occupied}(d') = \text{nil}$

eff:  $\text{loc}(r) \leftarrow d'$ ,  $\text{occupied}(d') = r$ ,  $\text{occupied}(d) = \text{nil}$

$\text{PSP}(\Sigma, \pi)$

loop

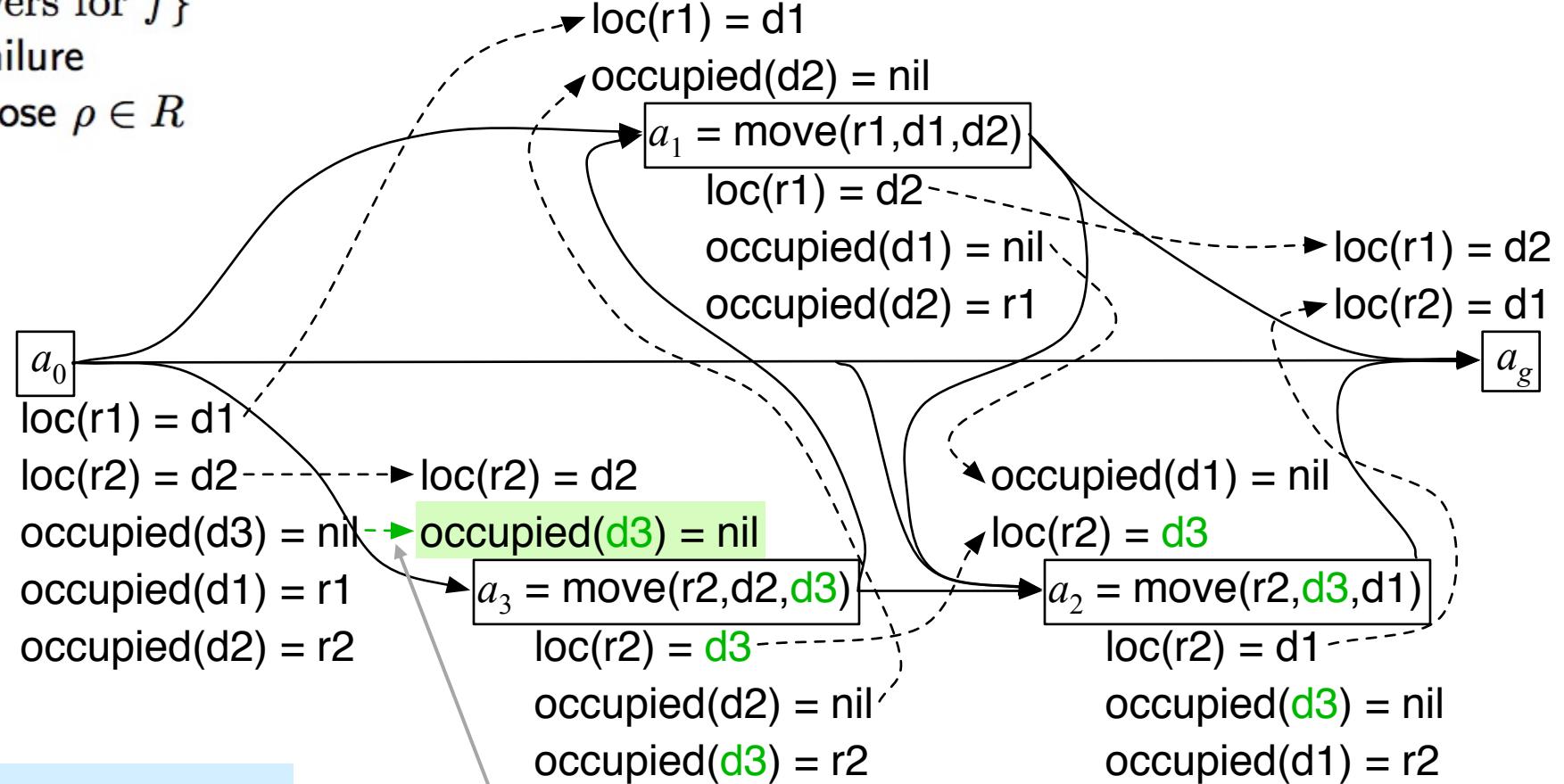
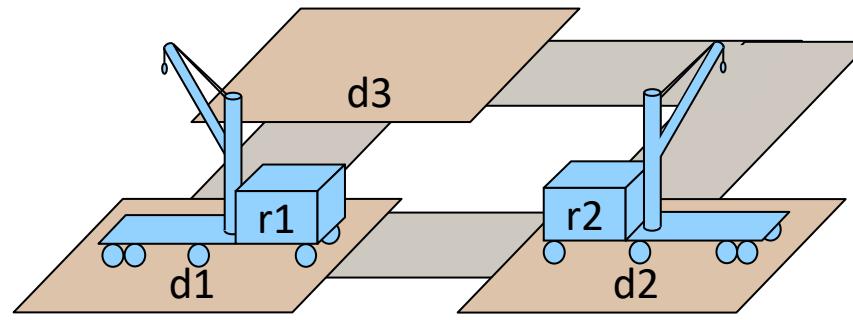
```

if  $\text{Flaws}(\pi) = \emptyset$  then return  $\pi$ 
arbitrarily select  $f \in \text{Flaws}(\pi)$ 
 $R \leftarrow \{\text{all feasible resolvers for } f\}$ 
if  $R = \emptyset$  then return failure
nondeterministically choose  $\rho \in R$ 
 $\pi \leftarrow \rho(\pi)$ 
return  $\pi$ 

```

- no open goals
- no threats
- we're done

# PSP Algorithm



$\text{move}(r, d, d')$

pre:  $\text{loc}(r) = d$ ,  $\text{occupied}(d') = \text{nil}$

eff:  $\text{loc}(r) \leftarrow d'$ ,  $\text{occupied}(d') = r$ ,  $\text{occupied}(d) = \text{nil}$

*causal link from  $a_0$  with substitution  $d' \leftarrow d3$*

$$\text{PSP}(\Sigma, \pi)$$

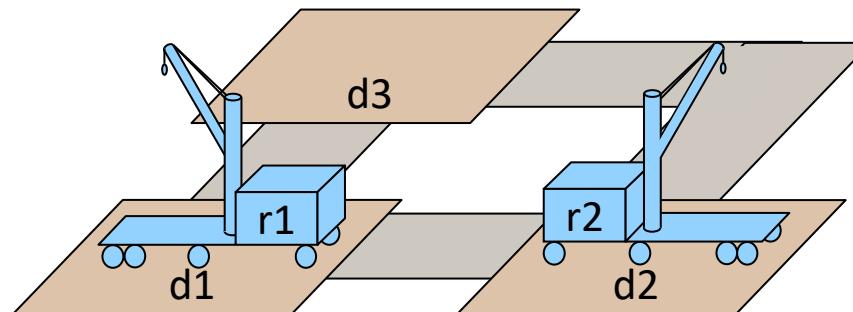
loop

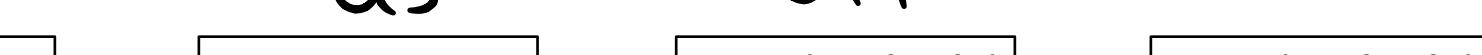
```

if  $Flaws(\pi) = \emptyset$  then return  $\pi$ 
arbitrarily select  $f \in Flaws(\pi)$ 
 $R \leftarrow \{\text{all feasible resolvers for } f\}$ 
if  $R = \emptyset$  then return failure
nondeterministically choose  $\rho \in R$ 
 $\pi \leftarrow \rho(\pi)$ 
turn  $\pi$ 

```

# PSP Algorithm



- $\pi \leftarrow \rho(\pi)$
  - return  $\pi$
  - The solution we found: 
  - Another: 
  - Infinitely many others

**move( $r$ ,  $d$ ,  $d'$ )**

pre:  $\text{loc}(r) = d$ ,  $\text{occupied}(d') = \text{nil}$

**eff:**  $\text{loc}(r) \leftarrow d'$ ,  $\text{occupied}(d') = r$ ,  $\text{occupied}(d) = \text{nil}$

$\text{PSP}(\Sigma, \pi)$

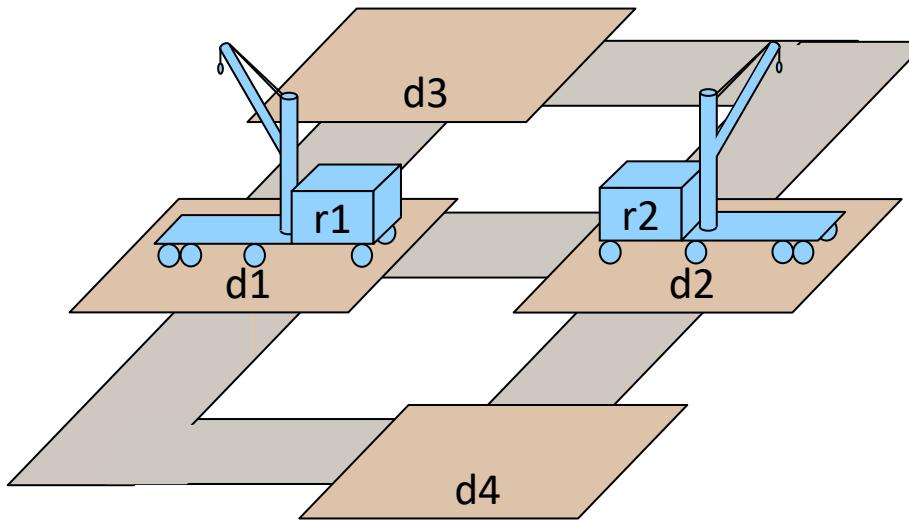
loop

```

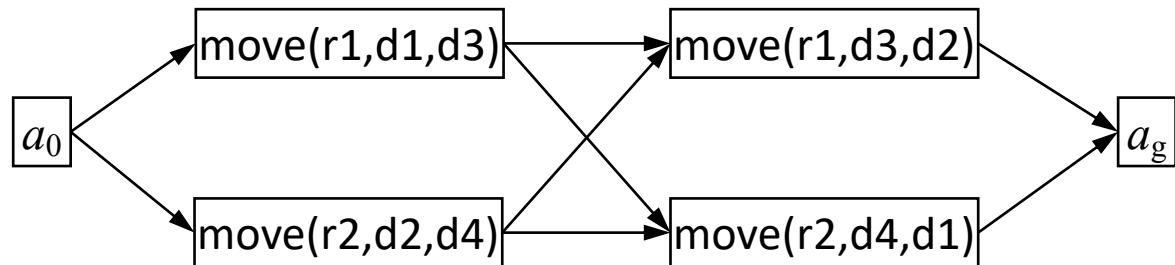
if  $\text{Flaws}(\pi) = \emptyset$  then return  $\pi$ 
arbitrarily select  $f \in \text{Flaws}(\pi)$ 
 $R \leftarrow \{\text{all feasible resolvers for } f\}$ 
if  $R = \emptyset$  then return failure
nondeterministically choose  $\rho \in R$ 
 $\pi \leftarrow \rho(\pi)$ 
return  $\pi$ 

```

# PSP Algorithm



- Add another location to the initial state
- Still have all of the solutions on the previous page
- This time, PSP also can return partially-ordered solutions



$\text{move}(r, d, d')$

pre:  $\text{loc}(r) = d$ ,  $\text{occupied}(d') = \text{nil}$

eff:  $\text{loc}(r) \leftarrow d'$ ,  $\text{occupied}(d') = r$ ,  $\text{occupied}(d) = \text{nil}$

$\text{PSP}(\Sigma, \pi)$

loop

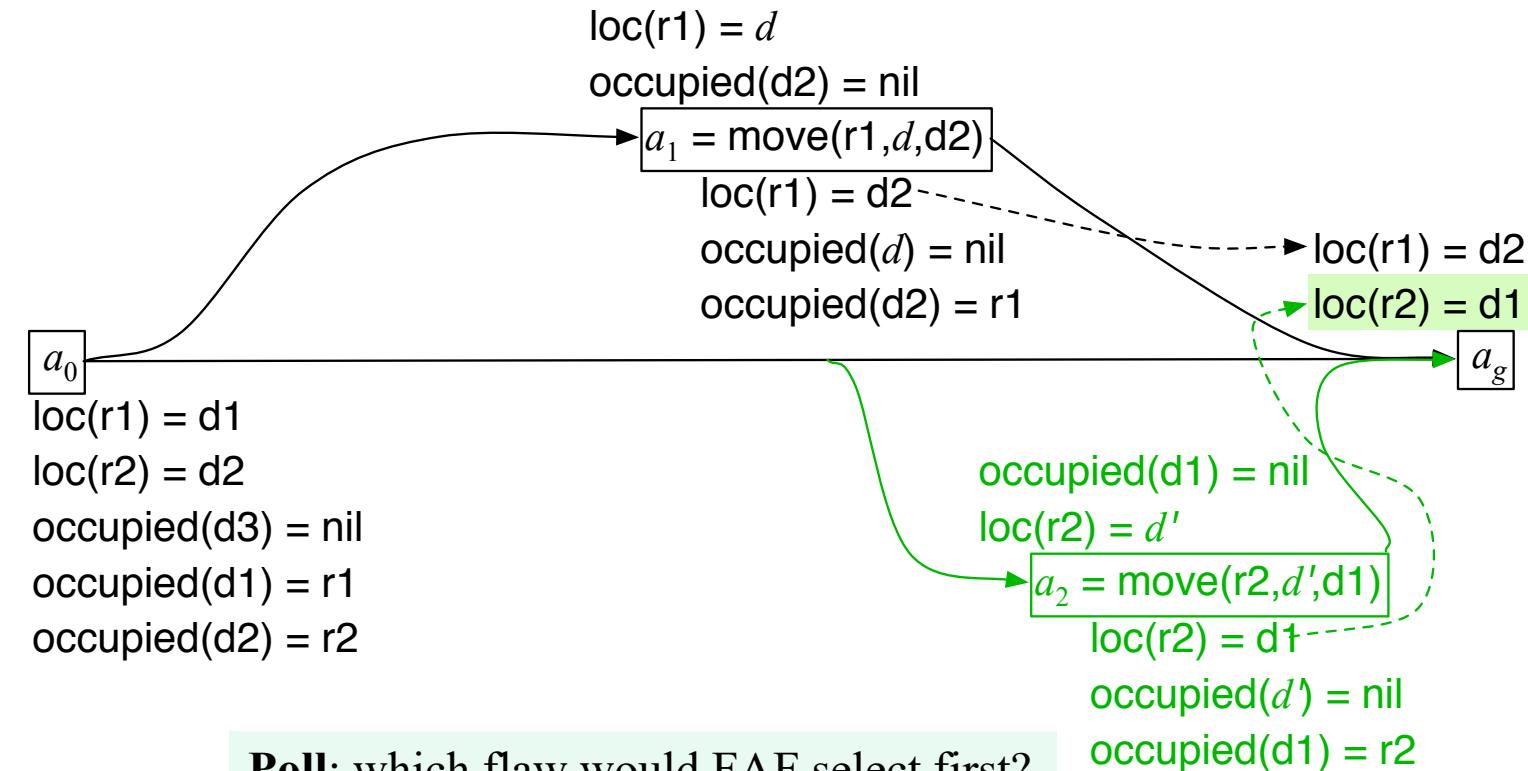
```

    if  $\text{Flaws}(\pi) = \emptyset$  then return  $\pi$ 
    arbitrarily select  $f \in \text{Flaws}(\pi)$ 
     $R \leftarrow \{\text{all feasible resolvers for } f\}$ 
    if  $R = \emptyset$  then return failure
    nondeterministically choose  $\rho \in R$ 
     $\pi \leftarrow \rho(\pi)$ 
return  $\pi$ 

```

- Selecting a flaw to resolve in PSP  $\approx$  selecting a variable to instantiate in a CSP
  - ▶ AND-branch in both cases
- **Fewest Alternatives First (FAF):**
  - ▶ select flaw with fewest resolvers  
 $\approx$  Minimum Remaining Values (MRV) heuristic for CSPs

## Selecting a Flaw



**Poll:** which flaw would FAF select first?

- $\text{loc}(r1)=d$
- $\text{occupied}(d2) = \text{nil}$
- $\text{occupied}(d2) = \text{nil}$
- $\text{loc}(r2)=d'$
- no preference

$\text{move}(r, d, d')$

pre:  $\text{loc}(r) = d$ ,  $\text{occupied}(d') = \text{nil}$

eff:  $\text{loc}(r) \leftarrow d'$ ,  $\text{occupied}(d') = r$ ,  $\text{occupied}(d) = \text{nil}$

$\text{PSP}(\Sigma, \pi)$

loop

```

if  $\text{Flaws}(\pi) = \emptyset$  then return  $\pi$ 
arbitrarily select  $f \in \text{Flaws}(\pi)$ 
 $R \leftarrow \{\text{all feasible resolvers for } f\}$ 
if  $R = \emptyset$  then return failure
nondeterministically choose  $\rho \in R$ 
 $\pi \leftarrow \rho(\pi)$ 
return  $\pi$ 

```

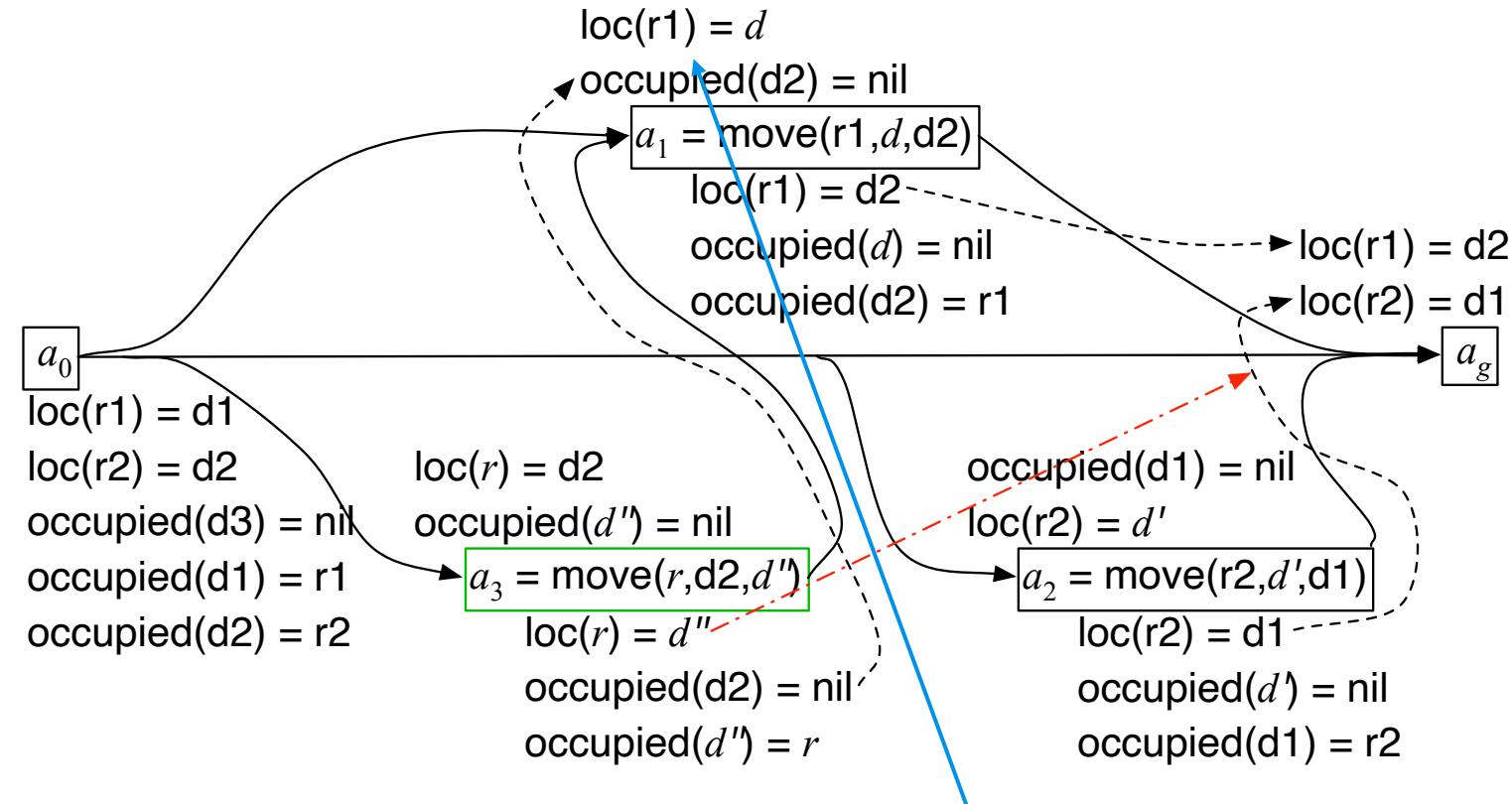
- Choosing a resolver for a flaw  $\approx$  assigning a value to a variable in a CSP
  - In both cases, an OR-branch
- Least Constraining Resolver (LCR):*
  - prefer resolver that rules out the fewest resolvers for the other flaws
  $\approx$  Least Constraining Value (LCV) heuristic for CSPs

$\text{move}(r, d, d')$

pre:  $\text{loc}(r) = d$ ,  $\text{occupied}(d') = \text{nil}$

eff:  $\text{loc}(r) \leftarrow d'$ ,  $\text{occupied}(d') = r$ ,  $\text{occupied}(d) = \text{nil}$

# Choosing a Resolver



**Poll:** for  $\text{loc}(r_1)=d$  in  $a_1$ , which resolver would LCR choose first?

- causal link from a new action
- causal link from  $a_0$ , with substitution  $d \leftarrow d_1$
- causal link from  $a_3$ , with substitutions  $r \leftarrow r_1$ ,  $d'' \leftarrow d$
- no preference

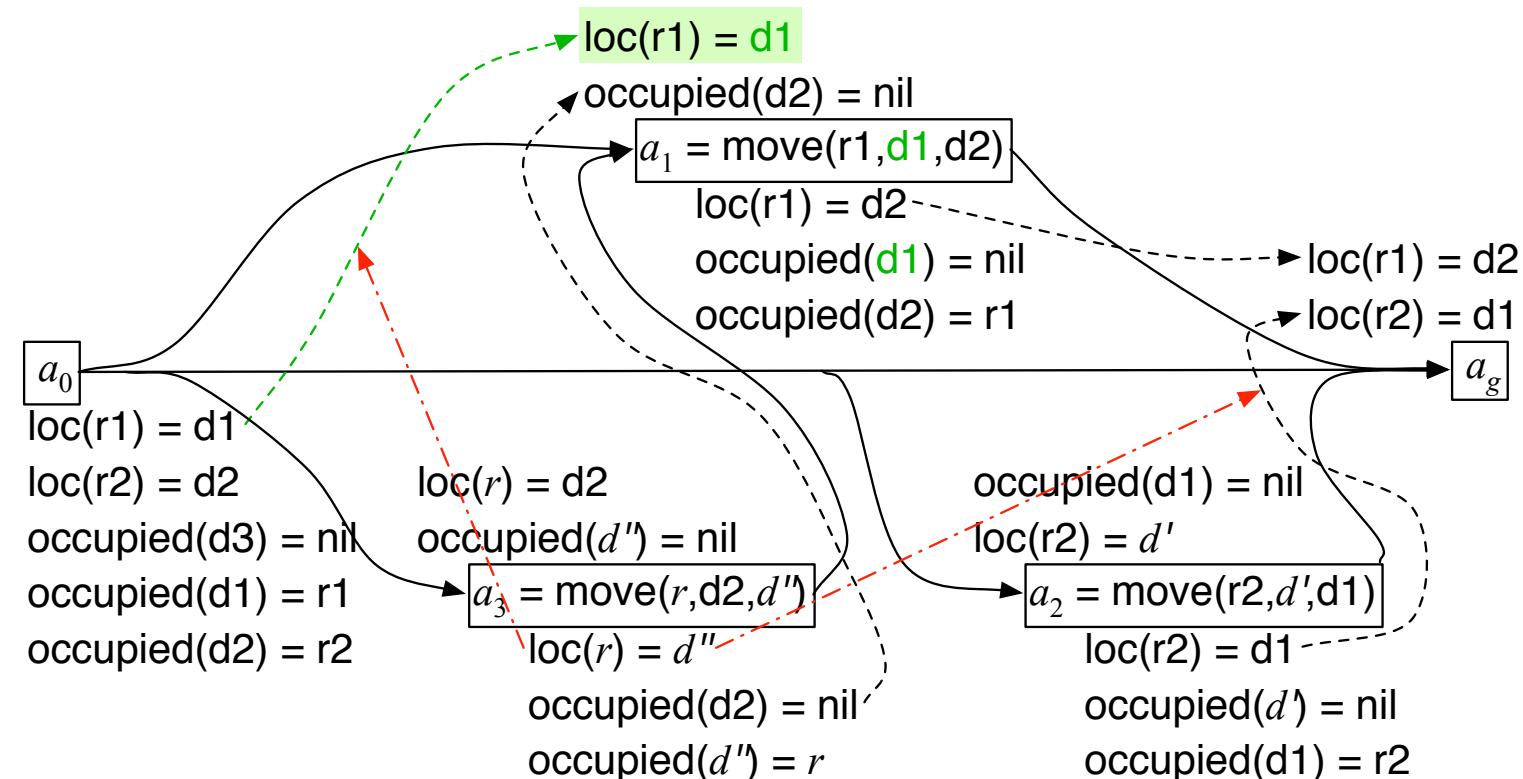
PSP( $\Sigma, \pi$ )

loop

- if  $Flaws(\pi) = \emptyset$  then return  $\pi$
- arbitrarily select  $f \in Flaws(\pi)$
- $R \leftarrow \{\text{all feasible resolvers for } f\}$
- if  $R = \emptyset$  then return failure
- nondeterministically choose  $\rho \in R$
- $\pi \leftarrow \rho(\pi)$
- return  $\pi$

- Least Constraining Resolver (LCR):
  - ▶ prefer resolver that rules out the fewest resolvers for the other flaws
  - ≈ Least Constraining Value (LCV) heuristic for CSPs
- Problem (in PSP but not in CSPs):
  - ▶ LCR can keep adding new actions forever

# Choosing a Resolver



Perhaps this might work:

- **Avoid New Actions (ANA) heuristic:**
  - ▶ prefer resolvers that don't add new actions
  - ▶ use LCR as tie-breaker

PSP( $\Sigma, \pi$ )

loop

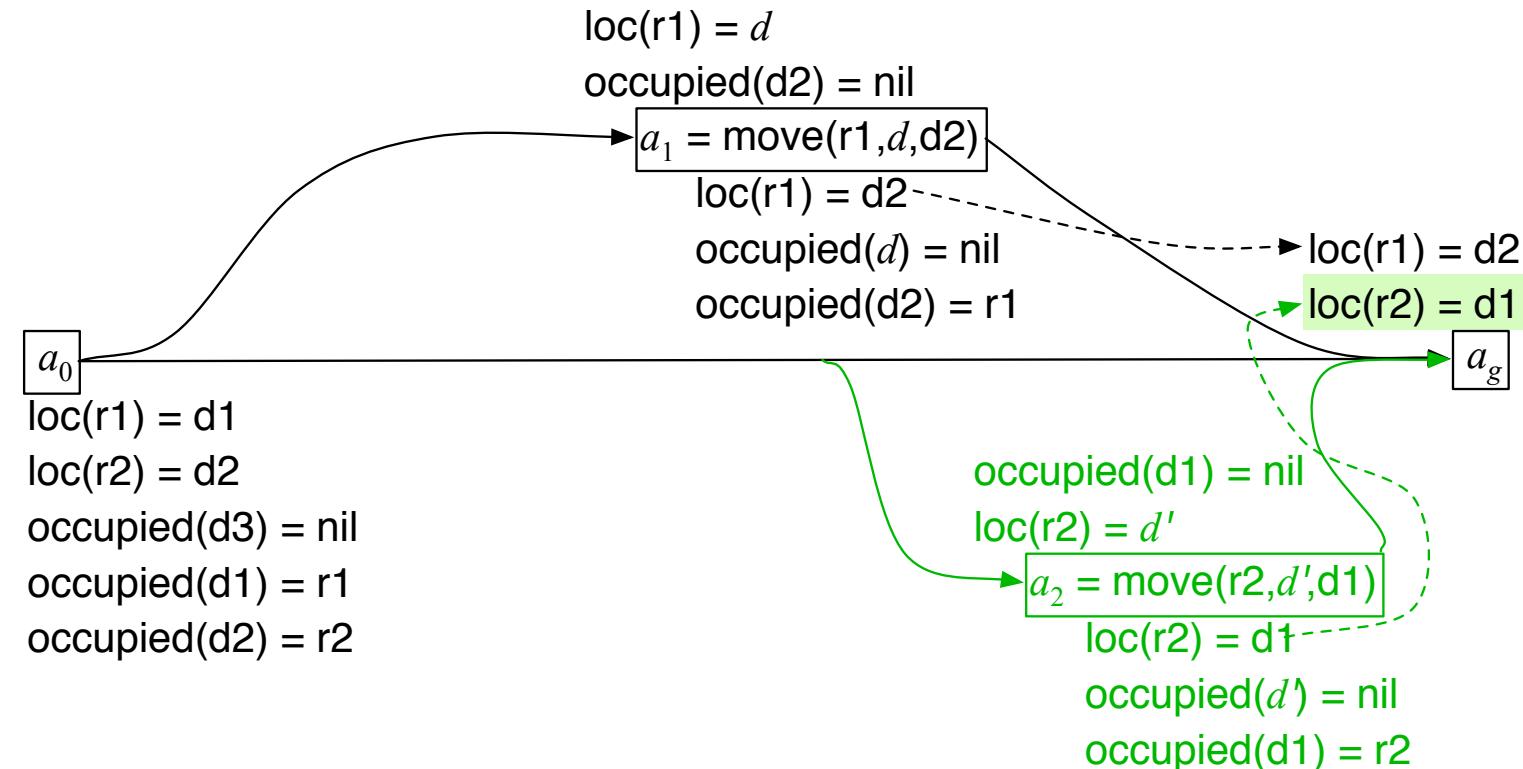
- if  $Flaws(\pi) = \emptyset$  then return  $\pi$
- arbitrarily select  $f \in Flaws(\pi)$
- $R \leftarrow \{\text{all feasible resolvers for } f\}$
- if  $R = \emptyset$  then return failure
- nondeterministically choose  $\rho \in R$
- $\pi \leftarrow \rho(\pi)$
- return  $\pi$

- ~~Least Constraining Resolver (LCR):~~
  - ▶ prefer resolver that rules out the fewest resolvers for the other flaws
  - ≈ Least Constraining Value (LCV) heuristic for CSPs
- Problem (in PSP but not in CSPs):
  - ▶ LCR can keep adding new actions forever

Perhaps this might work:

- *Avoid New Actions (ANA)* heuristic:
  - ▶ prefer resolvers that don't add new actions
  - ▶ use LCR as tie-breaker

# Choosing a Resolver

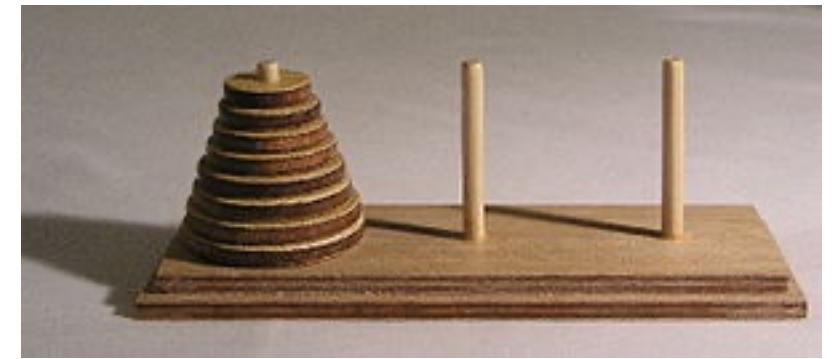


- Problem: ANA will prefer these two choices:
  - ▶ For  $loc(r1)=d$  in  $a_1$ , use action  $a_0$  with substitution  $d \leftarrow d1$
  - ▶ For  $loc(r2)=d'$  in  $a_2$ , use action  $a_0$  with substitution  $d' \leftarrow d2$
  - ▶  $a_1 = move(r1, d1, d2); a_2 = move(r2, d2, d1)$ 
    - Makes the problem unsolvable  $\Rightarrow$  need to backtrack
- Perhaps use ANA anyway?

# Discussion

- Problem: how to prune infinitely long paths in the search space?
  - ▶ Loop detection is based on recognizing states or goals we've seen before
  - ▶ Partially ordered plan: don't know the states
- Prune if  $\pi$  contains the same *action* more than once?
  - $\langle a_1, a_2, \dots, a_1, \dots \rangle$
  - ▶ No. Sometimes need the same action again in another state
    - e.g., Towers of Hanoi: move disk1 from peg1 to peg2
- Weak pruning technique
  - ▶ Prune all partial plans that contain more than  $|S|$  actions
  - ▶ Not very helpful
- I don't know whether there's a better pruning technique

$$\dots \longrightarrow s \longrightarrow s' \longrightarrow s$$



# Summary

- 2.5 Plan-Space Search
  - ▶ Definitions
    - Partially ordered plans and solutions
    - partial plans
    - causal links
  - ▶ flaws:
    - open goals
    - threats
  - ▶ resolvers
  - ▶ PSP algorithm
    - long example
    - brief discussion of node-selection heuristics, pruning techniques