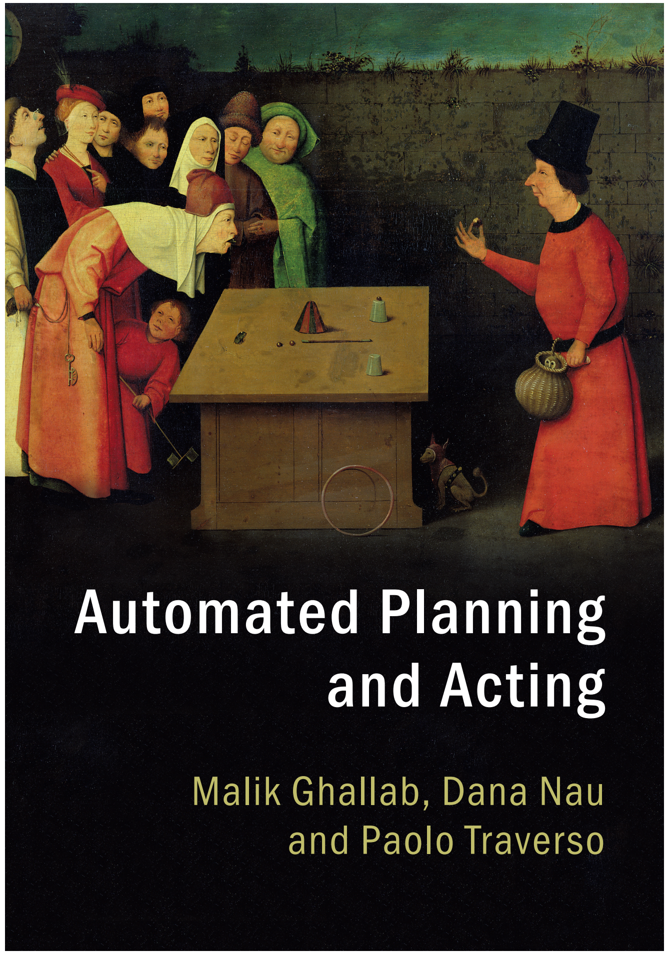
Last update: February 22, 2022

Chapter 2

Deliberation with Deterministic Models

2.4: Backward Search
2.5: Plan-Space Search

Dana S. Nau
University of Maryland



http://www.laas.fr/planning

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):

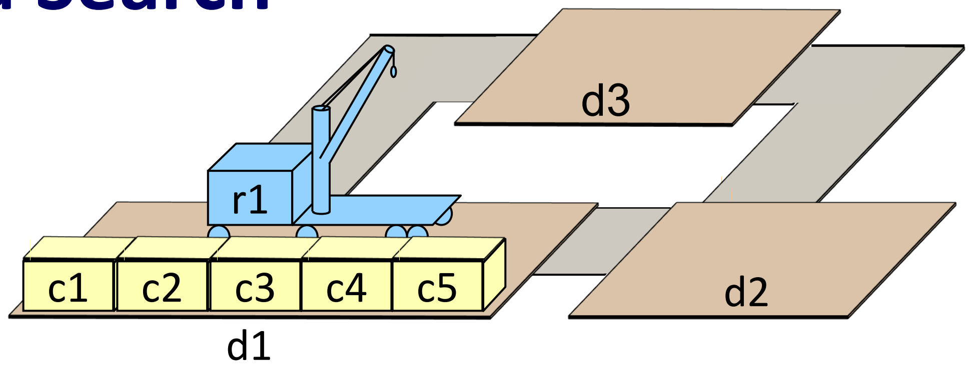
- *Next* ⟶ 2.4 Backward search
- 2.5 Plan-space search

Additional slides:

- 2.7.8 LTL_planning.pdf

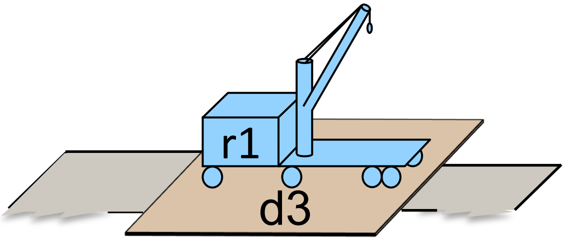
2.4 Backward Search

- 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 = {all states that satisfy g} then
  - $S_{g'}$ = {all states s such that $\gamma(s,a) \in S_g$}



s_0 = {loc(c1)=d1, loc(c2)=d1, …}

g = {loc(r1)=d3}

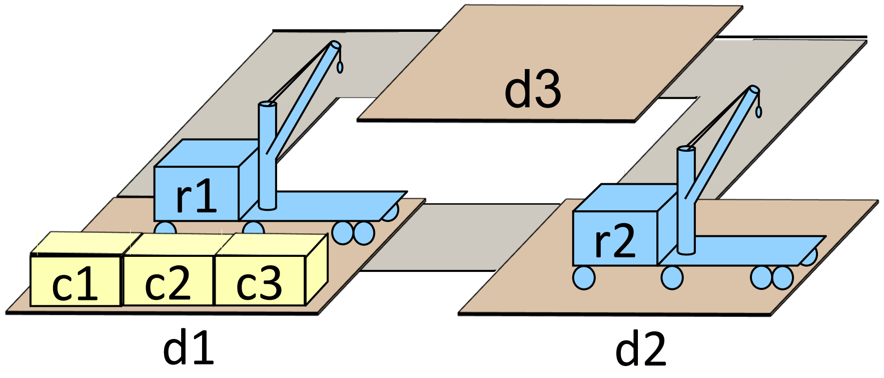


- Forward: 7 applicable actions
  - five load actions, two move actions
- Backward: g = {loc(r1)=d3}
  - two relevant actions: move(r1,d1,d3), move(r1,d2,d3)

Relevance

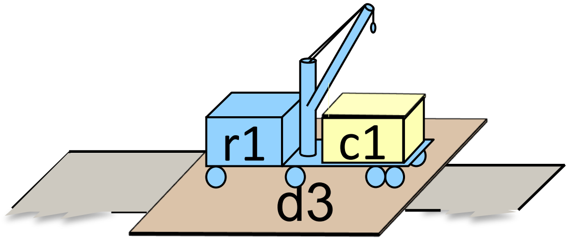
- Idea: when can a be useful as the last action of a plan to achieve g?
  - a makes at least one atom in g true that wasn't true already
  - a doesn't make any part of g false

- a is *relevant* for $g = \{x_1{=}c_1, x_2{=}c_2, \ldots, x_k{=}c_k\}$ if
  - at least one atom in g is also in eff(a)
    - e.g., if eff(a) contains $x_1 \leftarrow c_1$
  - eff(a) doesn't make any atom in g false
    - e.g., eff(a) must not contain $x_2 \leftarrow c_2'$ (where $c_2' \neq c_2$)
  - whenever pre(a) requires an atom of g to be false, eff(a) makes the atom true
    - e.g., if pre(a) contains $x_3 = c_3'$ (where $c_3' \neq c_3$), then eff(a) must contain $x_3 \leftarrow c_3$



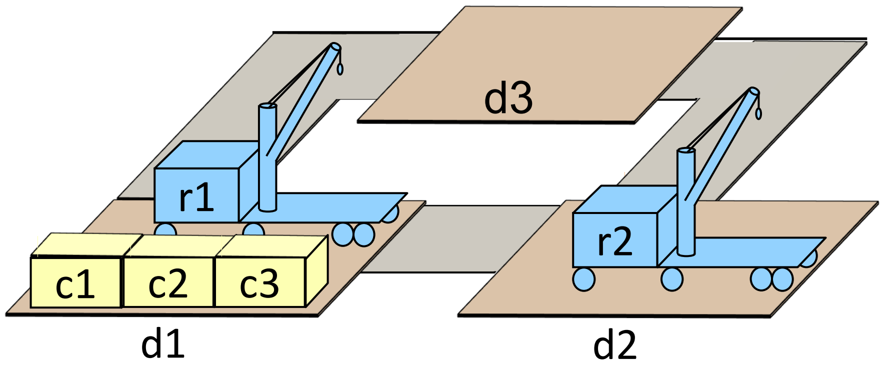
s = {loc(c1)=d1, loc(c2)=d1, loc(c3)=d1,
loc(r1)=d2, cargo(r1)=nil,
loc(r2)=d2, cargo(r2)=nil}

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



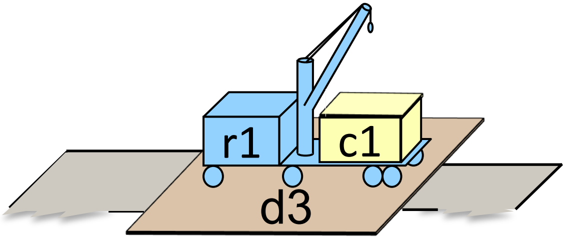
g = {cargo(r1)=c1, loc(r1)=d3}

Relevance



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

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



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

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

load(r,c,l)
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) $\leftarrow l$

Range(r) = *Robots* = {r1,r2}

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

Range(c) = *Containers* = {c1,c2,c3}

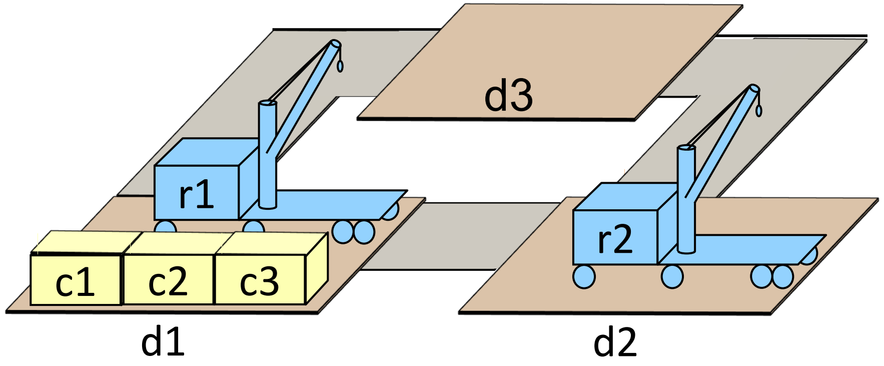
Poll: for each action below, is it relevant for g?

load(r1,c1,d1) load(r1,c1,d2) put(r2,c1,d3)

move(r1,d1,d3) move(r1,d3,d1) 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 $\gamma^{-1}(g, \text{load(r2,c1,d1)})$?



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

load(r,c,l)
    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) $\leftarrow l$

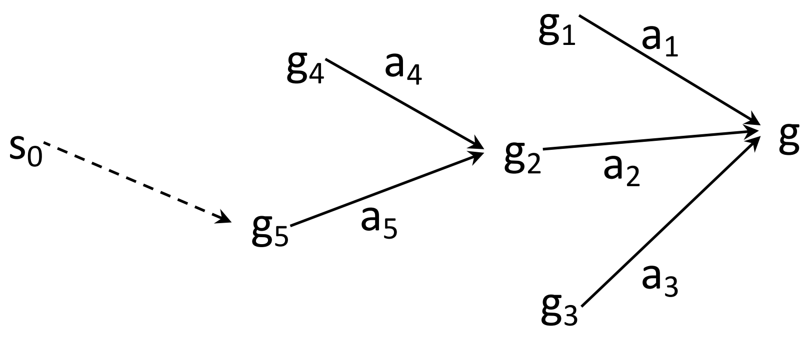
Range(r) = *Robots*

Range(l) = Range(m) = *Locs*

Range(c) = *Containers*

Backward Search

Backward-search(Σ, s_0, g_0)
    $\pi \leftarrow \langle\rangle$; $g \leftarrow g_0$ (i)
    loop
        if s_0 satisfies g then return π
        $A' \leftarrow \{a \in A \mid a$ is relevant for $g\}$
        if $A' = \varnothing$ 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* $\cup \{g\}$

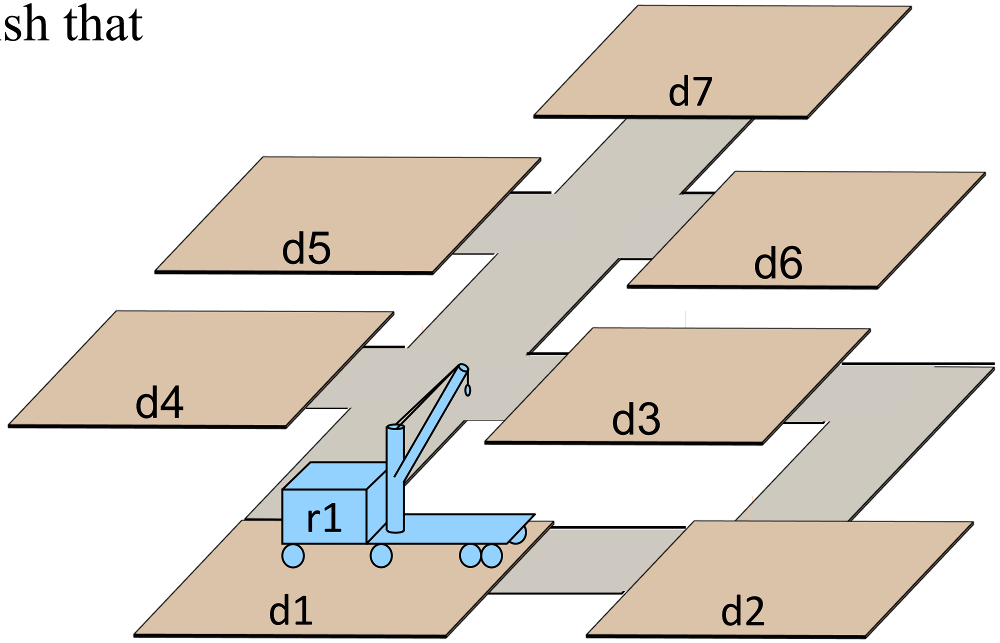
  or this:

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

- With cycle checking, sound and complete
    - If (Σ, 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



move(r1,d1,d3)

move(r1,d2,d3)

move(r1,d4,d3)

⋮

move(r1,d7,d3)

γ^{-1} $g = \{$loc(r1)=d3$\}$

move(r1,y,d3) $\xleftarrow{\gamma^{-1}}$ $g = \{$loc(r1)=d3$\}$

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

```
Backward-search(Σ, s_0, g_0)
  π ← ⟨⟩;  g ← g_0
  loop
    if s_0 satisfies g then return π
    A' ← {a ∈ A | a is relevant for g}
    if A' = ∅ then return failure
    nondeterministically choose a ∈ A'
    g ← γ^{-1}(g, a)
    π ← a.π
```

For classical planning, this can be simplified

```
Lifted-backward-search(𝒜, s_0, g)
  π ← the empty plan
  loop
    if s_0 satisfies g then return π
    A ← {(o, θ)|o is a standardization of an action template in 𝒜,
              θ is an mgu for an atom of g and an atom of effects⁺(o),
              and γ^{-1}(θ(g), θ(o)) is defined}
    if A = ∅ then return failure
    nondeterministically choose a pair (o, θ) ∈ A
    π ← the concatenation of θ(o) and θ(π)
    g ← γ^{-1}(θ(g), θ(o))
```

Summary

- 2.4 Backward State-Space Search
  - Relevance, γ^{-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)

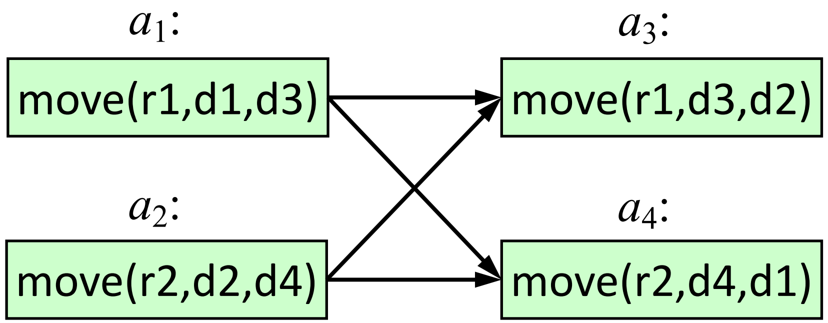
- Basic idea:
  - Backward search from the goal
  - Each node of the search space is a *partial plan* that contains *flaws*
    - Remove the flaws by making *refinements*
  - If successful, we'll get a *partially ordered* solution

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 π such that every total ordering of π is a solution for P

Poll. Let P be the planning problem at right, and π be the partially ordered plan below. Is π a partially ordered solution for P?

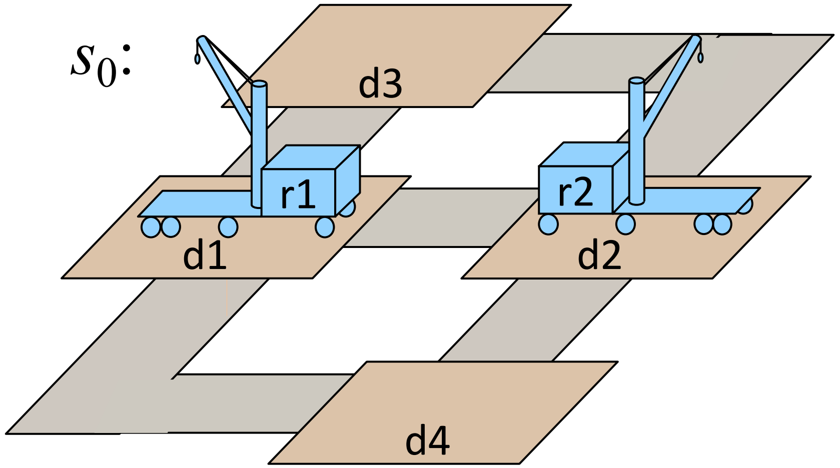
precedence constraints:
$a_1 \prec a_3,\ a_1 \prec a_4,$
$a_2 \prec a_3,\ a_2 \prec a_4$



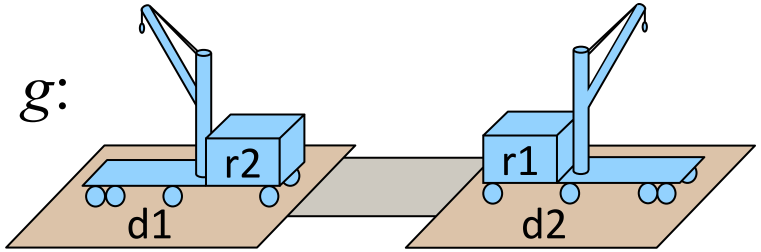
move(r, d, d')
pre: loc(r) = d, occupied(d') = nil
eff: loc(r) ← d', occupied(d') ← r, occupied(d) ← nil

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

s_0:

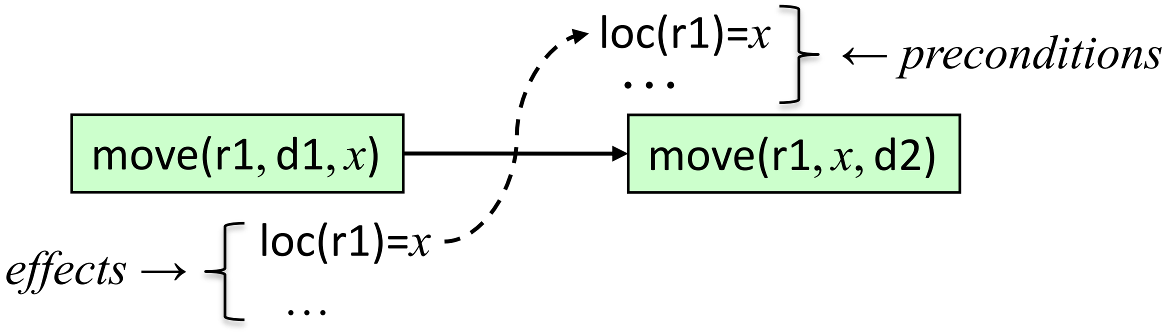


g:



Definitions

- *Partial plan*
  - partially ordered set of nodes that contain *partially instantiated* actions
  - *inequality constraints*, e.g. $z \neq x$ or $w \neq$ p1
  - *causal links* (dashed arcs)
    - constraint: action *a must* be the action that establishes action *b*'s precondition *p*

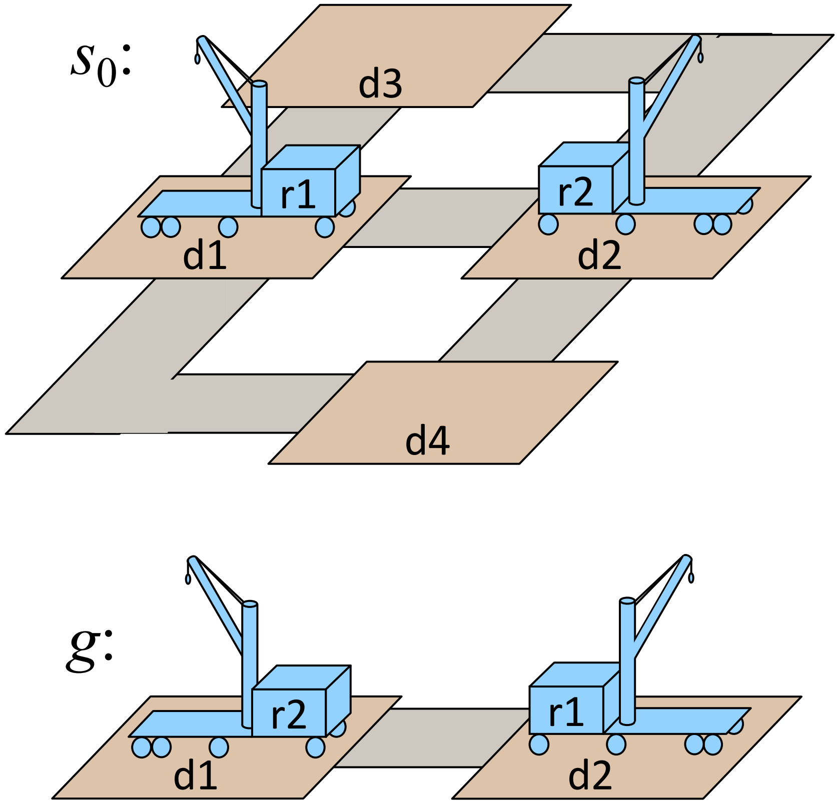


move(r, d, d')

pre: loc(r) = d, occupied(d') = nil

eff: loc(r) ← d', occupied(d') ← r, occupied(d) ← nil

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



Flaws: 1. Open Goals

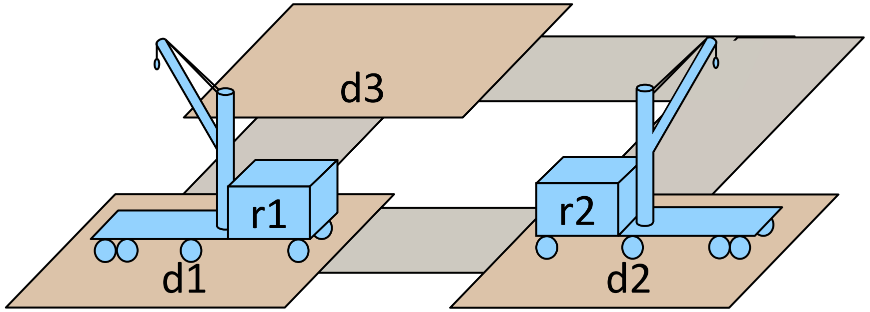
move(r, d, d')

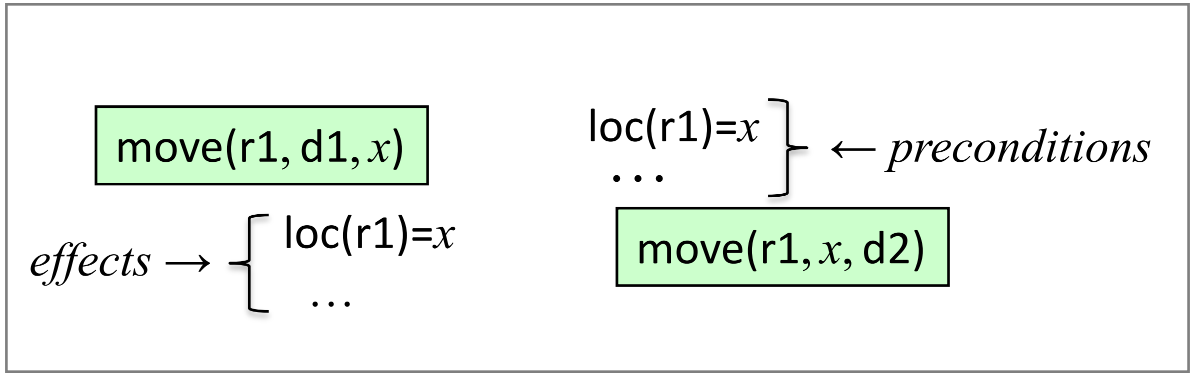
pre: loc(r) = d, occupied(d') = nil

eff: loc(r) $\leftarrow d'$, occupied(d') $\leftarrow r$, occupied(d) $\leftarrow$ nil

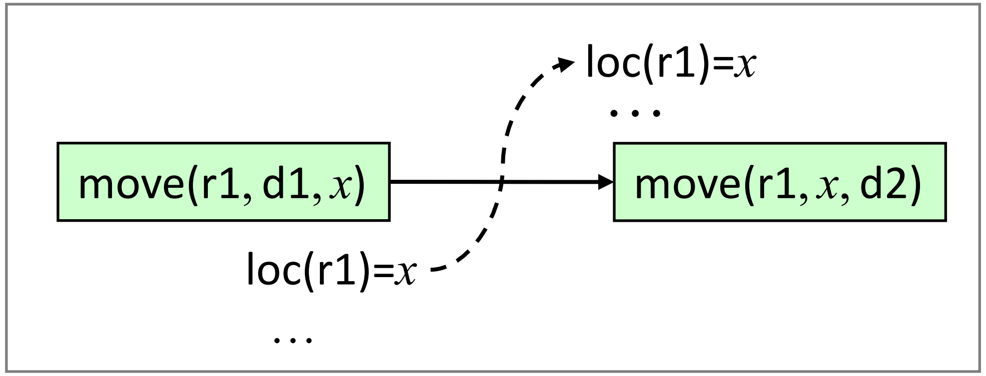
Range(r) = *Robots*
Range(d) = Range(d') = *Docks*

- 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 π, or add it to π) 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





substitute
$r \leftarrow$ r1

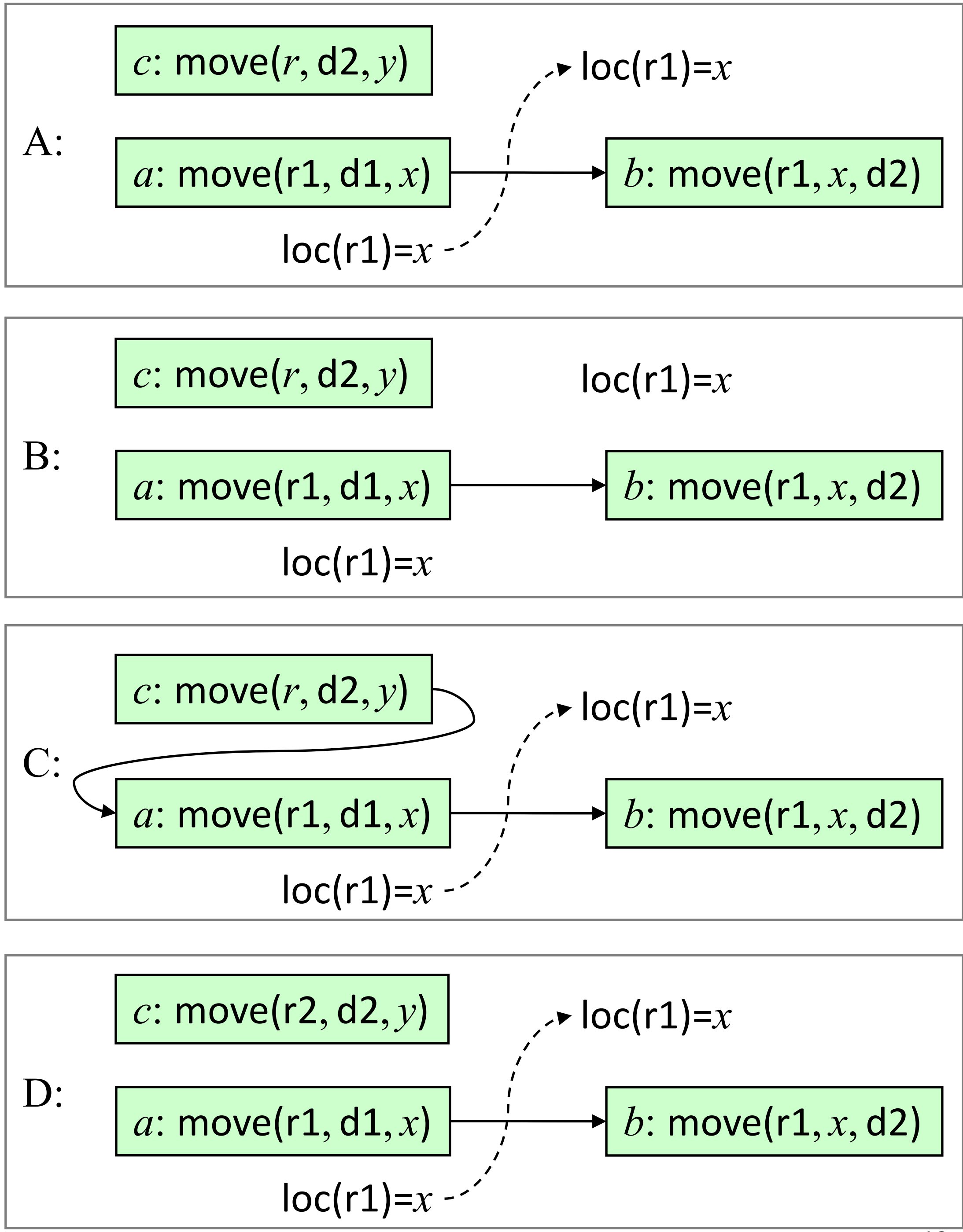


Flaws: 2. Threats

- 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?

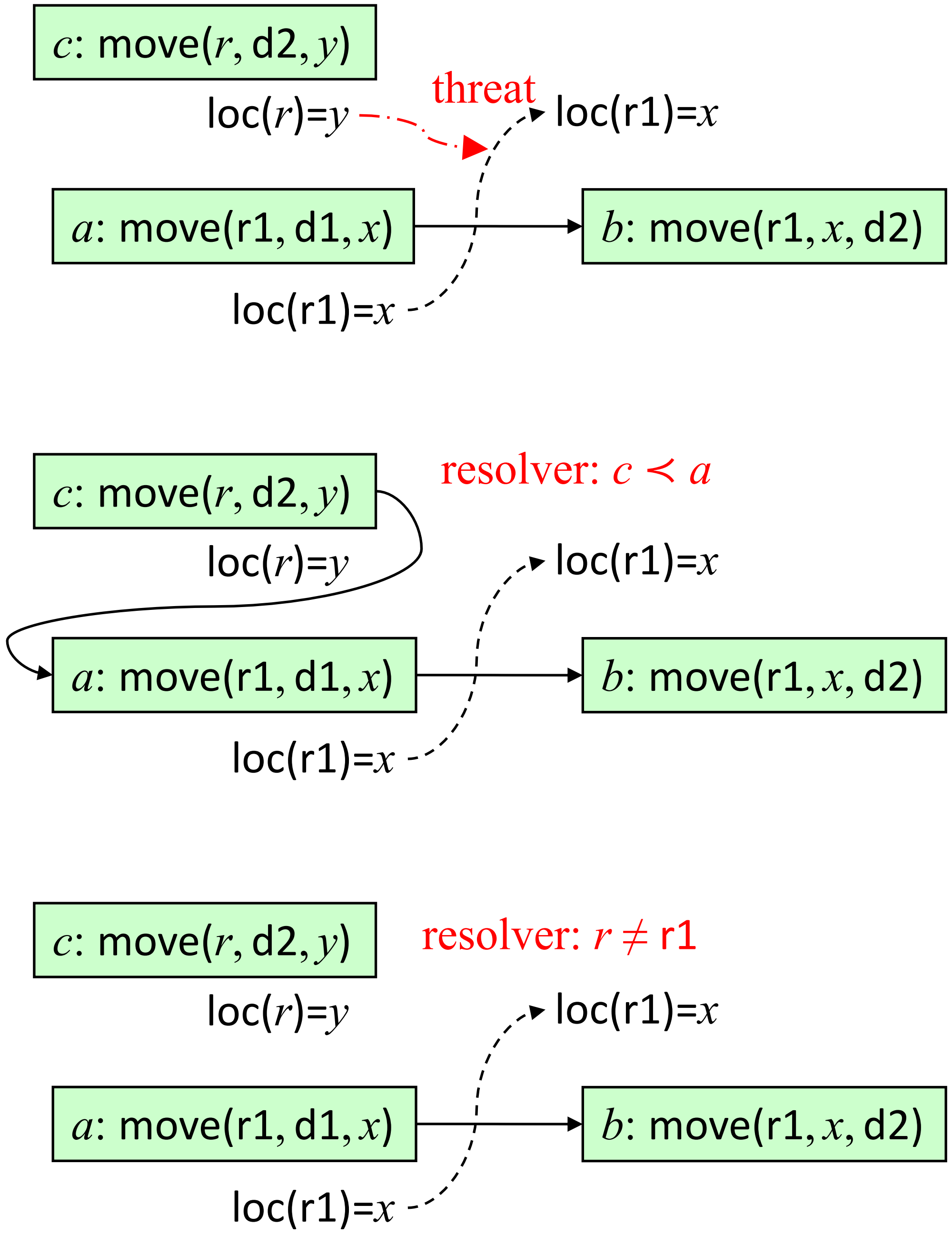


Resolving Threats

- Suppose action c threatens a causal link l from an effect of action a to a precondition p of action b

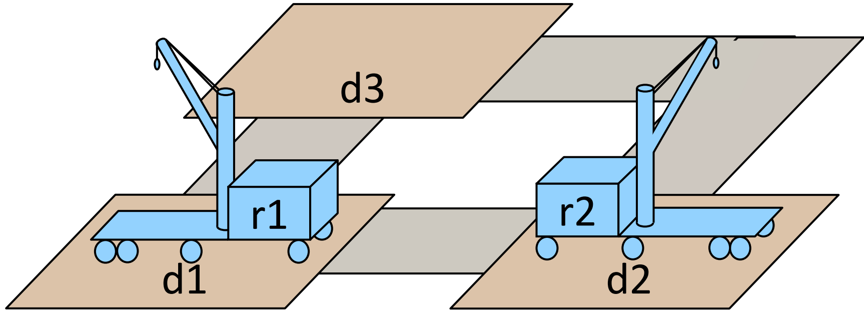
- Three possible resolvers:
  1. Add a precedence constraint $c \prec a$
  2. Add a precedence constraint $b \prec c$
  3. Add inequality constraints that prevent c from affecting p

- Each of these is applicable iff it doesn't make the plan inconsistent
  - e.g., 2 isn't applicable if the plan already has $c \prec b$



PSP Algorithm

PSP(Σ, π)
    loop
        if $Flaws(\pi) = \varnothing$ then return π
        arbitrarily select $f \in Flaws(\pi)$
        $R \leftarrow$ {all feasible resolvers for f}
        if $R = \varnothing$ then return failure
        nondeterministically choose $\rho \in R$
        $\pi \leftarrow \rho(\pi)$
    return π



- 2 open goals
- no threats

select $\longrightarrow$ loc(r1) = d2
loc(r2) = d1

$\boxed{a_0} \longrightarrow \boxed{a_g}$

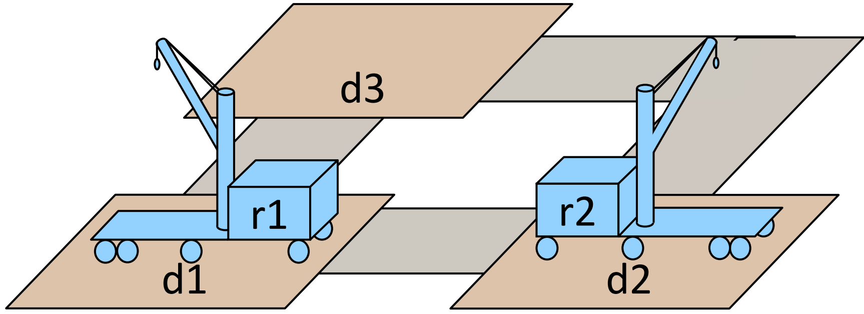
loc(r1) = d1
loc(r2) = d2
occupied(d3) = nil
occupied(d1) = r1
occupied(d2) = r2

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

PSP Algorithm

PSP(Σ, π)
  loop
    if $Flaws(\pi) = \varnothing$ then return π
    arbitrarily select $f \in Flaws(\pi)$
    $R \leftarrow$ {all feasible resolvers for f}
    if $R = \varnothing$ then return failure
    nondeterministically choose $\rho \in R$
    $\pi \leftarrow \rho(\pi)$
  return π

- 3 open goals
- no threats



loc(r1) = d
occupied(d2) = nil
a_1 = move(r1,d,d2)
loc(r1) = d2
occupied(d) = nil
occupied(d2) = r1

only resolver: causal link from a new action

a_0
loc(r1) = d1
loc(r2) = d2
occupied(d3) = nil
occupied(d1) = r1
occupied(d2) = r2

a_g
loc(r1) = d2
loc(r2) = d1

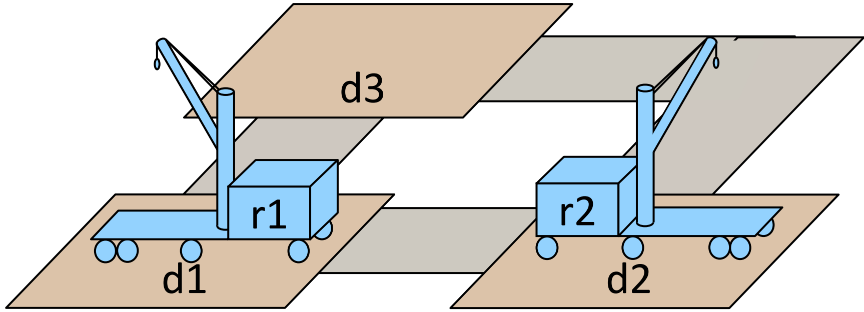
select

for every action a,
$a_0 \prec a \prec a_g$

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

Poll. Above, I said "only resolver". Is that correct?

PSP Algorithm



$\mathsf{PSP}(\Sigma, \pi)$
  loop
    if $\mathit{Flaws}(\pi) = \varnothing$ then return π
    arbitrarily select $f \in \mathit{Flaws}(\pi)$
    $R \leftarrow$ {all feasible resolvers for f}
    if $R = \varnothing$ then return failure
    nondeterministically choose $\rho \in R$
    $\pi \leftarrow \rho(\pi)$
  return π

- 4 open goals
- no threats

select →
loc(r1) = d
occupied(d2) = nil

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

only resolver: causal link from a new action

loc(r1) = d2
loc(r2) = d1

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

$\boxed{a_g}$

occupied(d1) = nil
loc(r2) = d'

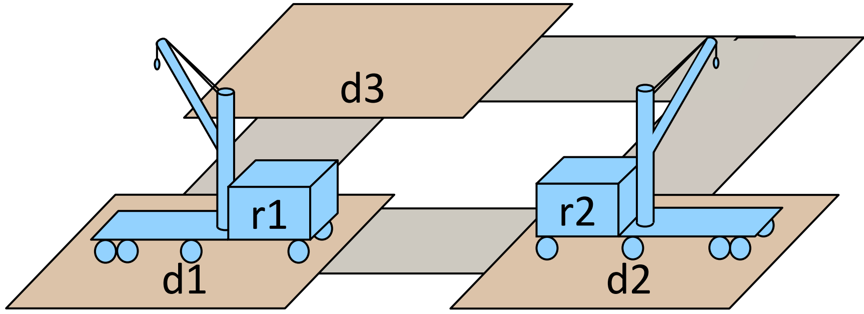
$\boxed{a_2 = \text{move(r2},d'\text{,d1)}}$
loc(r2) = d1
occupied(d') = nil
occupied(d1) = r2

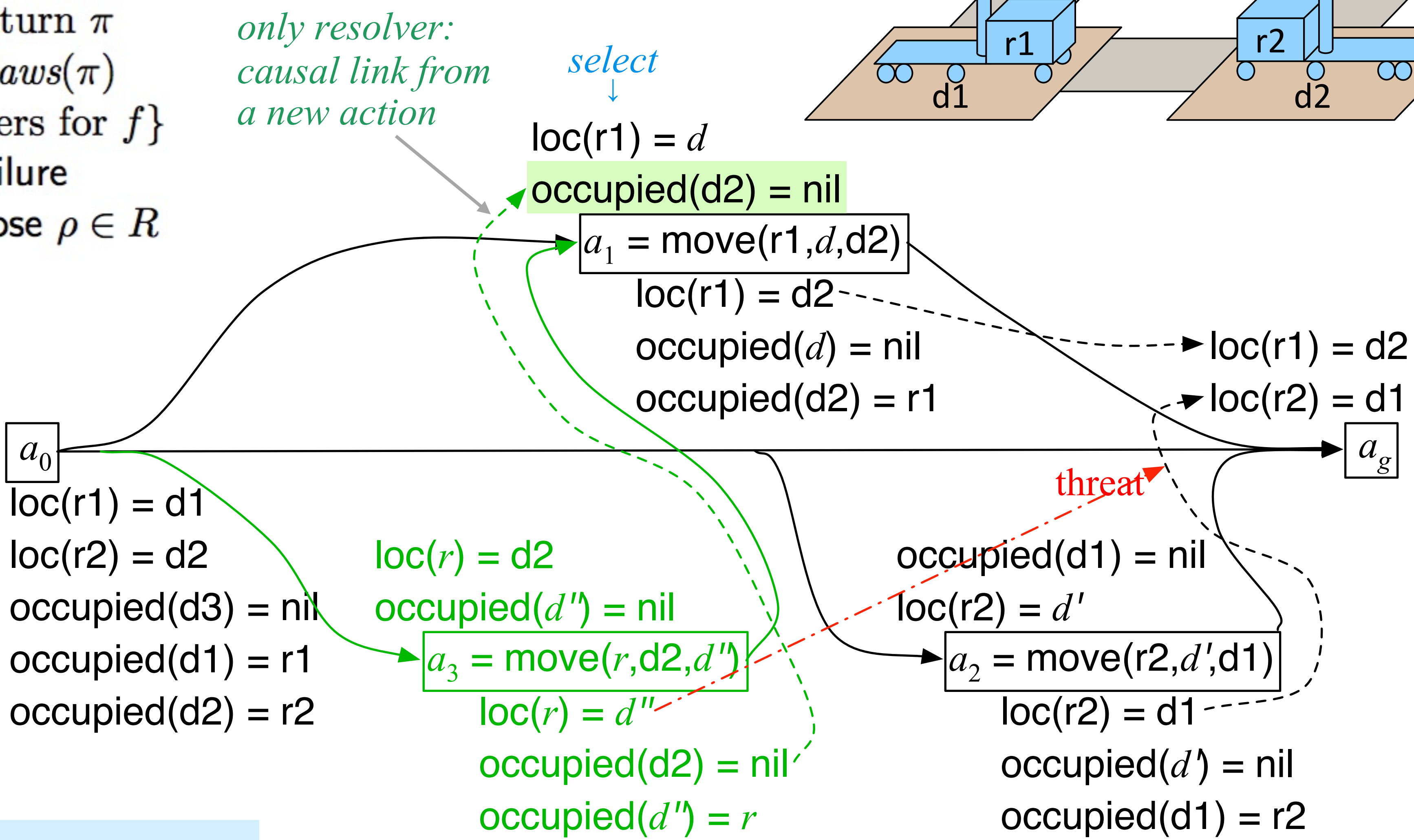
move(r, d, d')
pre: loc(r) = d, occupied(d') = nil
eff: loc(r) ← d', occupied(d') = r, occupied(d) = nil

PSP Algorithm

PSP(Σ, π)
    loop
        if $Flaws(\pi) = \varnothing$ then return π
        arbitrarily select $f \in Flaws(\pi)$
        $R \leftarrow$ {all feasible resolvers for f}
        if $R = \varnothing$ then return failure
        nondeterministically choose $\rho \in R$
        $\pi \leftarrow \rho(\pi)$
    return π

- 5 open goals
- 1 threat



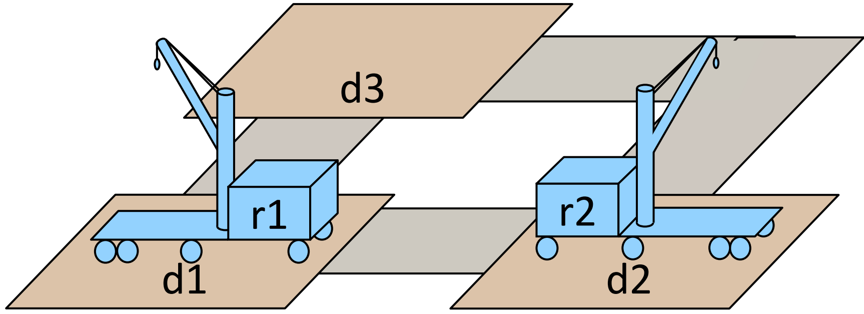


move(r, d, d')
pre: loc$(r) = d$, occupied(d') = nil
eff: loc$(r) \leftarrow d'$, occupied$(d') = r$, occupied(d) = nil

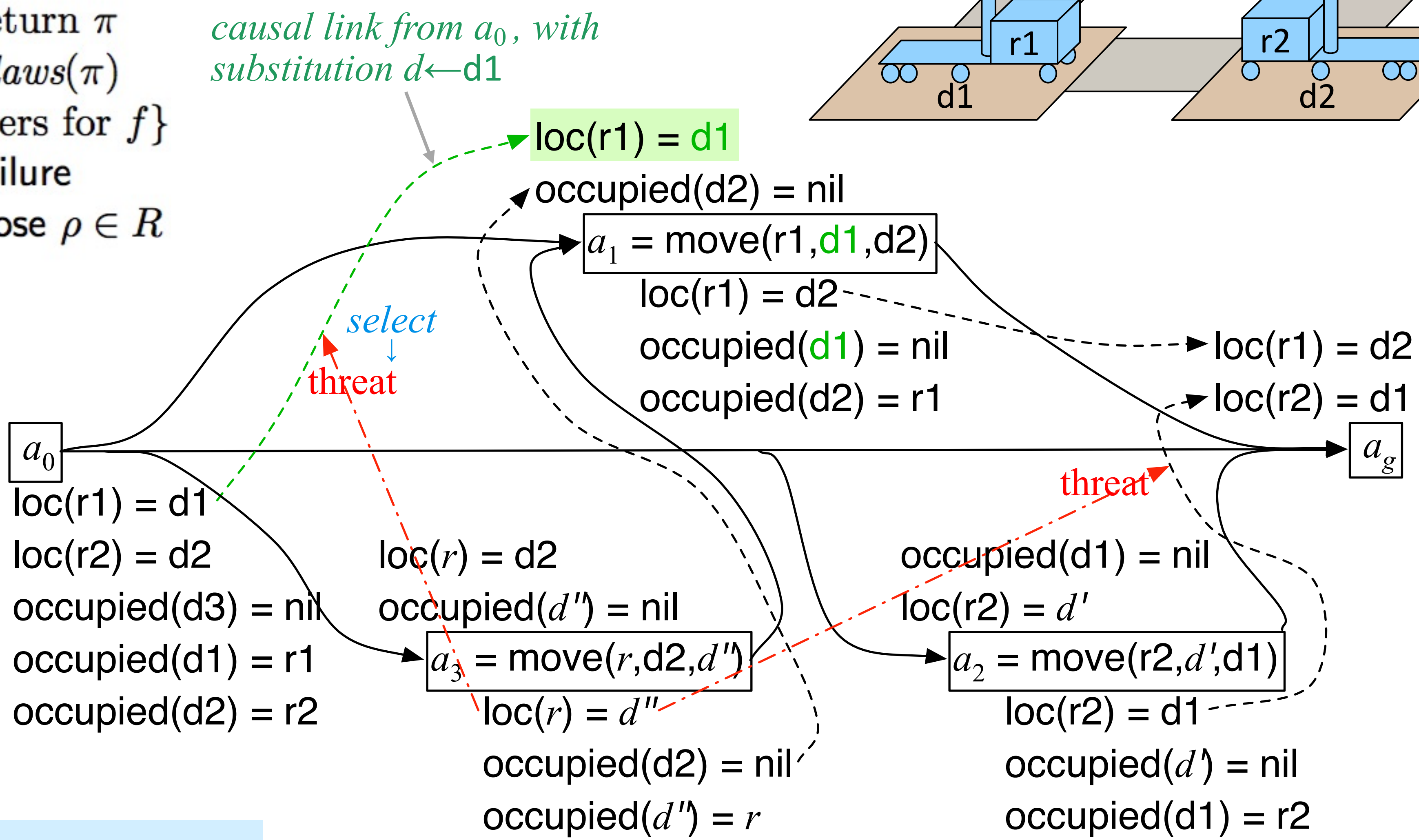
Poll: does a_3 threaten a_1's precondition loc(r1)=d?

PSP Algorithm

PSP(Σ, π)
  loop
    if $Flaws(\pi) = \varnothing$ then return π
    arbitrarily select $f \in Flaws(\pi)$
    $R \leftarrow$ {all feasible resolvers for f}
    if $R = \varnothing$ then return failure
    nondeterministically choose $\rho \in R$
    $\pi \leftarrow \rho(\pi)$
  return π



- 4 open goals
- 2 threats



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

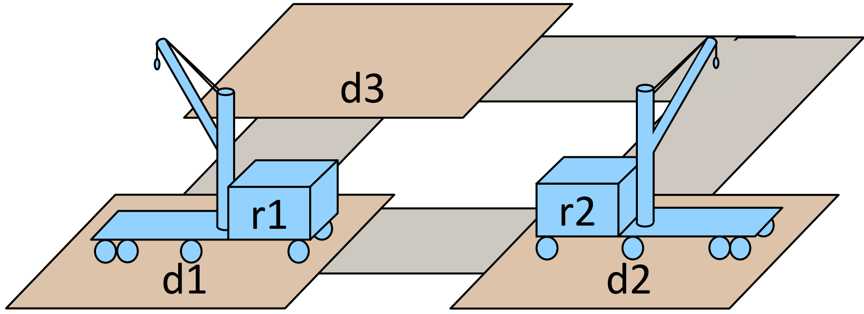
Poll: does a_3 threaten the causal link for a_g's precondition loc(r1)=d2?

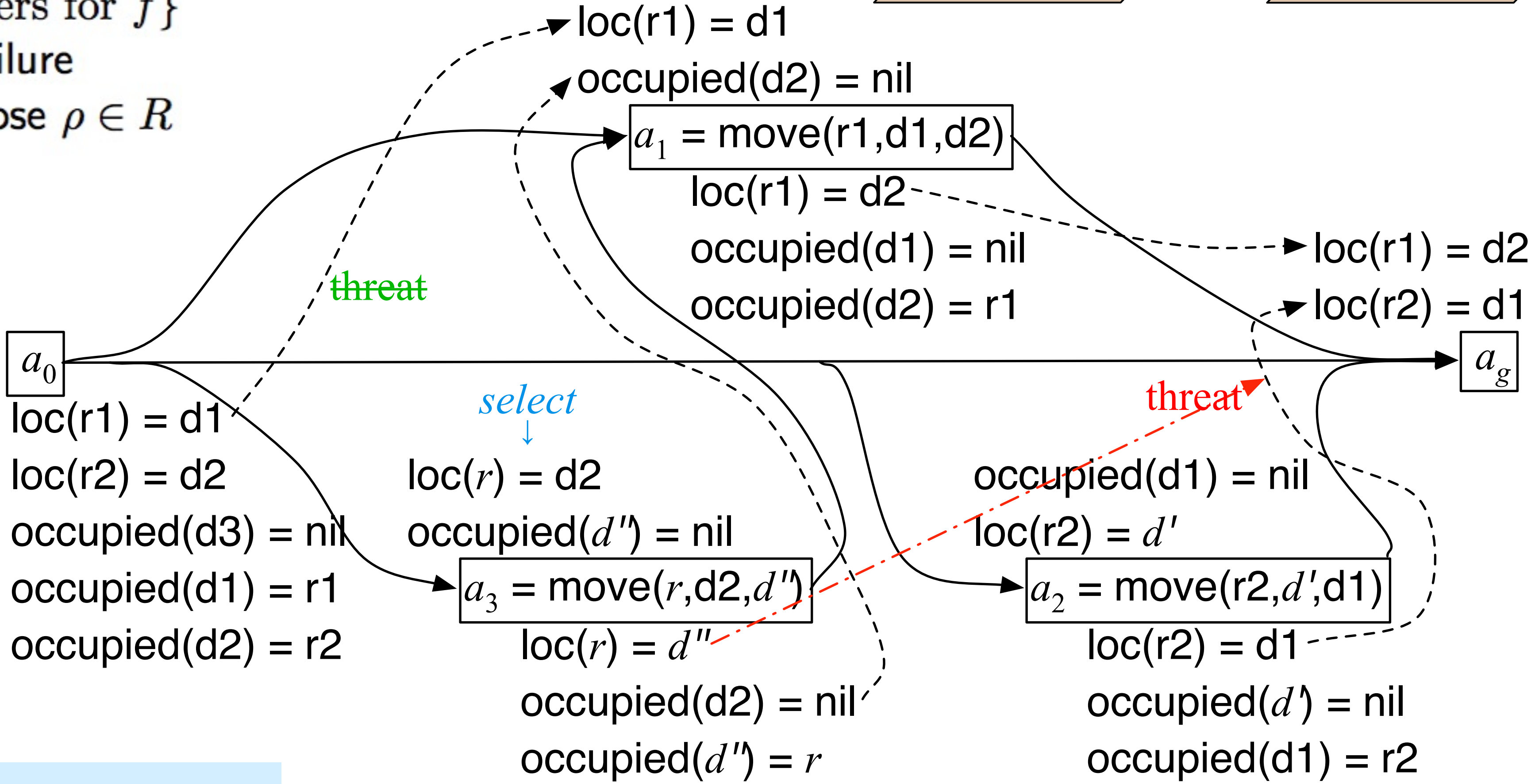
PSP Algorithm

PSP(Σ, π)
  loop
    if $Flaws(\pi) = \varnothing$ then return π
    arbitrarily select $f \in Flaws(\pi)$
    $R \leftarrow$ {all feasible resolvers for f}
    if $R = \varnothing$ then return failure
    nondeterministically choose $\rho \in R$
    $\pi \leftarrow \rho(\pi)$
  return π

- 4 open goals
- 1 threat

Constraint: $r \neq$ r1

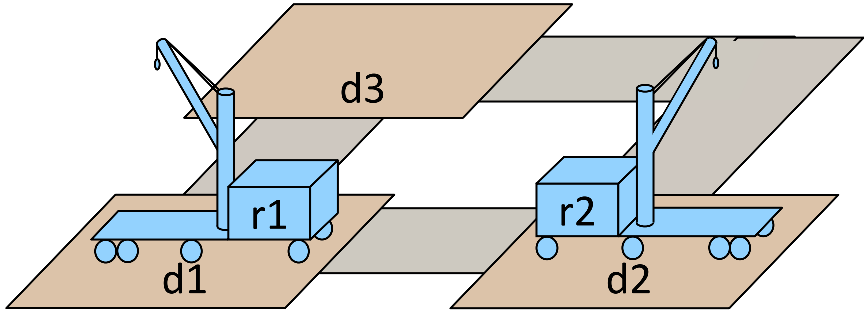




move(r, d, d')
pre: loc(r) = d, occupied(d') = nil
eff: loc(r) $\leftarrow$ d', occupied(d') = r, occupied(d) = nil

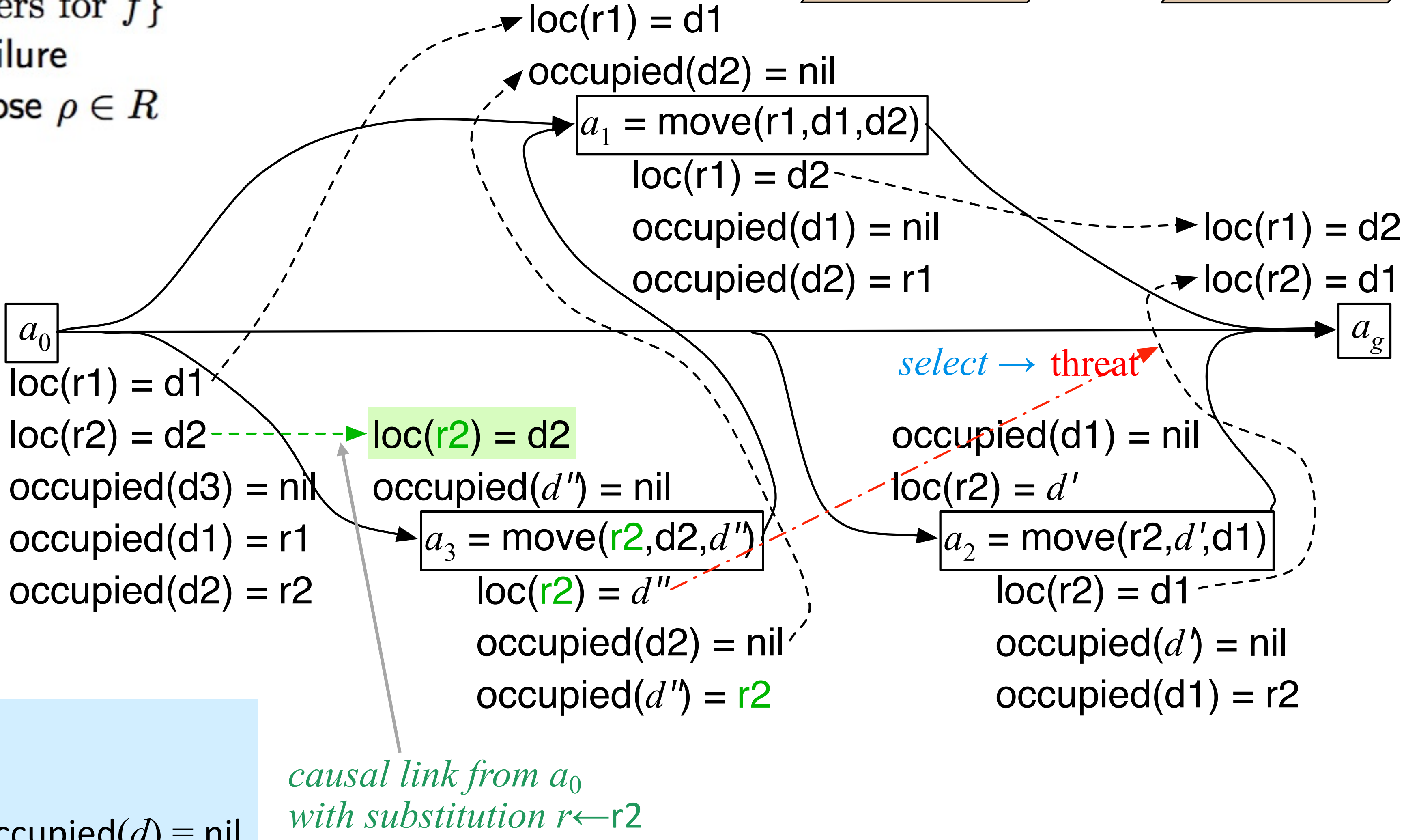
PSP Algorithm

PSP(Σ, π)
  loop
    if $Flaws(\pi) = \varnothing$ then return π
    arbitrarily select $f \in Flaws(\pi)$
    $R \leftarrow$ {all feasible resolvers for f}
    if $R = \varnothing$ then return failure
    nondeterministically choose $\rho \in R$
    $\pi \leftarrow \rho(\pi)$
  return π



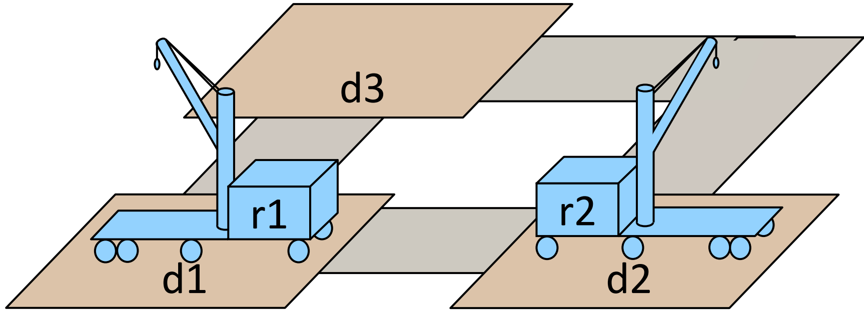
- 3 open goals
- 1 threat

~~Constraint: $r \neq$ r1~~



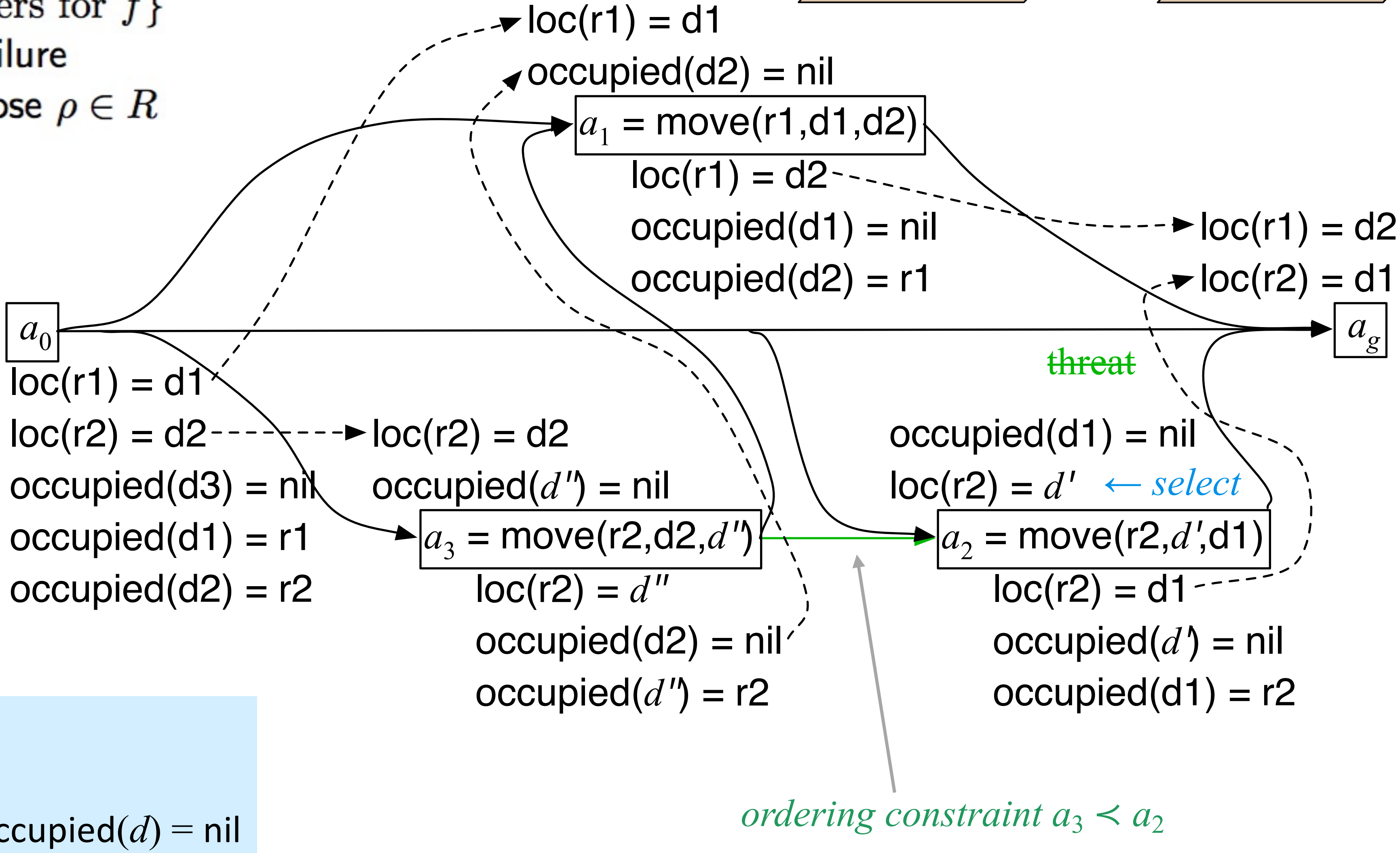
move(r, d, d')
pre: loc(r) = d, occupied(d') = nil
eff: loc(r) $\leftarrow d'$, occupied(d') = r, occupied(d) = nil

PSP Algorithm



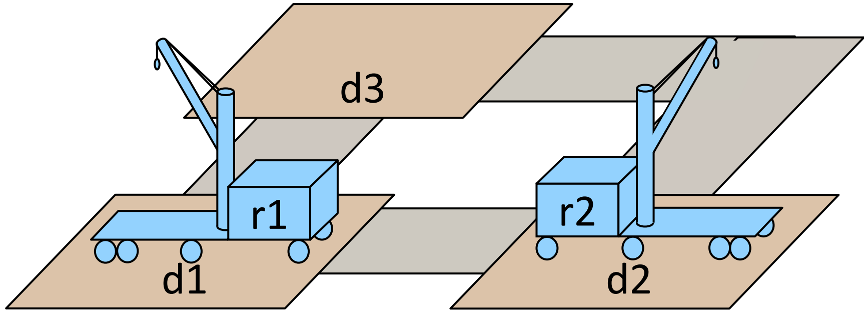
$\mathsf{PSP}(\Sigma, \pi)$
loop
    if $Flaws(\pi) = \varnothing$ then return π
    arbitrarily select $f \in Flaws(\pi)$
    $R \leftarrow$ {all feasible resolvers for f}
    if $R = \varnothing$ then return failure
    nondeterministically choose $\rho \in R$
    $\pi \leftarrow \rho(\pi)$
return π

- 3 open goals
- no threats



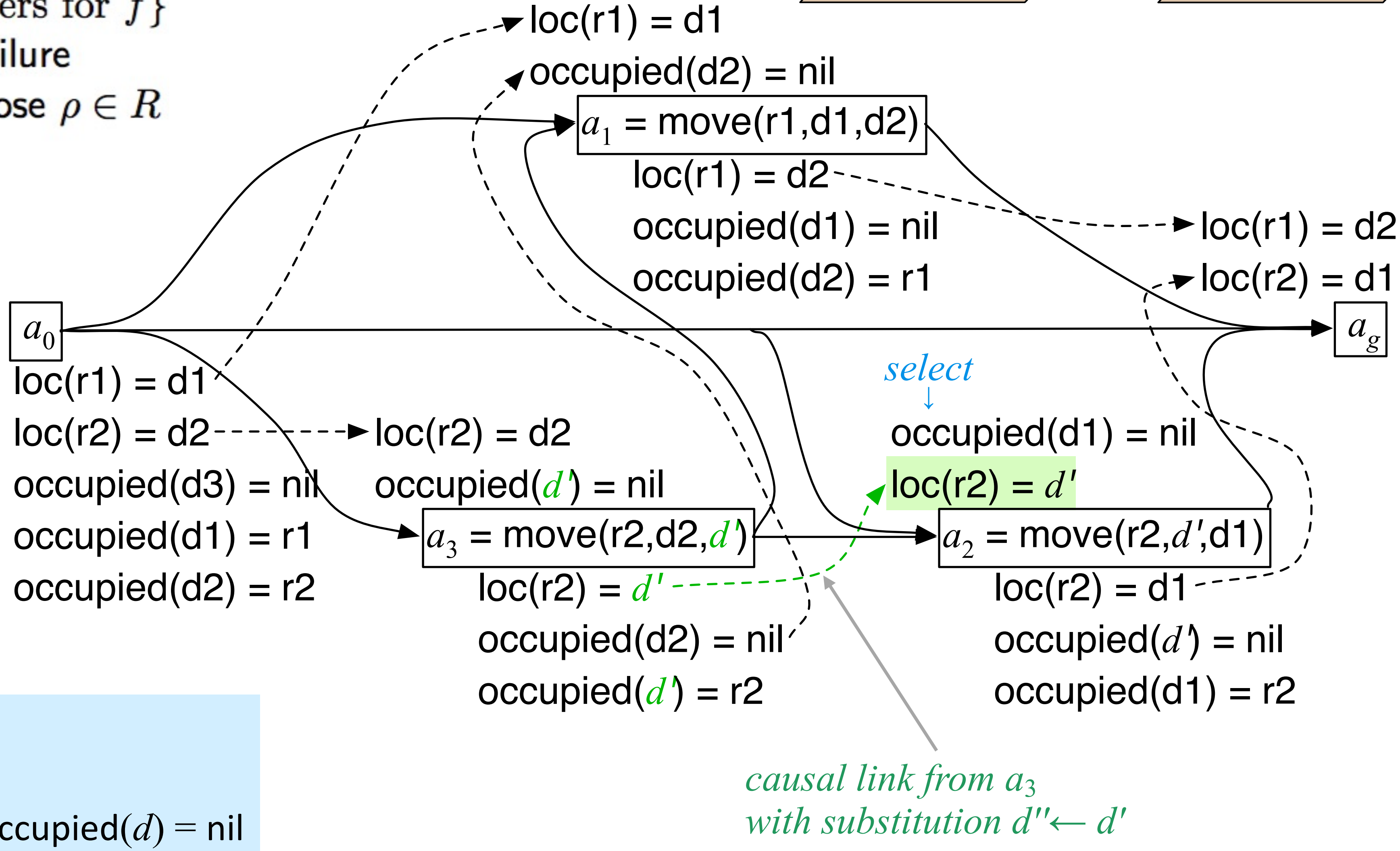
move(r, d, d')
pre: loc(r) = d, occupied(d') = nil
eff: loc(r) ← d', occupied(d') = r, occupied(d) = nil

PSP Algorithm



PSP(Σ, π)
  loop
    if $Flaws(\pi) = \varnothing$ then return π
    arbitrarily select $f \in Flaws(\pi)$
    $R \leftarrow$ {all feasible resolvers for f}
    if $R = \varnothing$ then return failure
    nondeterministically choose $\rho \in R$
    $\pi \leftarrow \rho(\pi)$
  return π

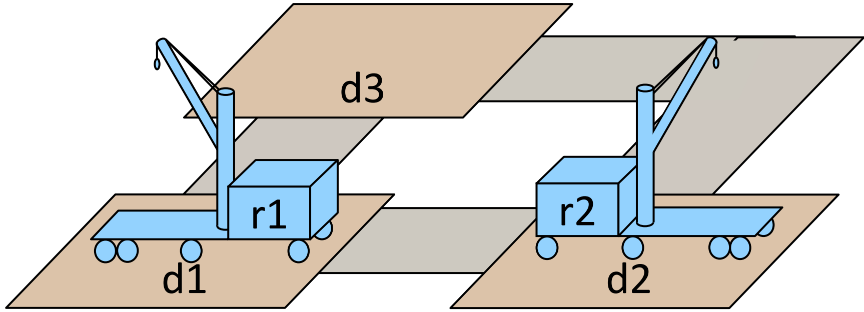
- 2 open goals
- no threats



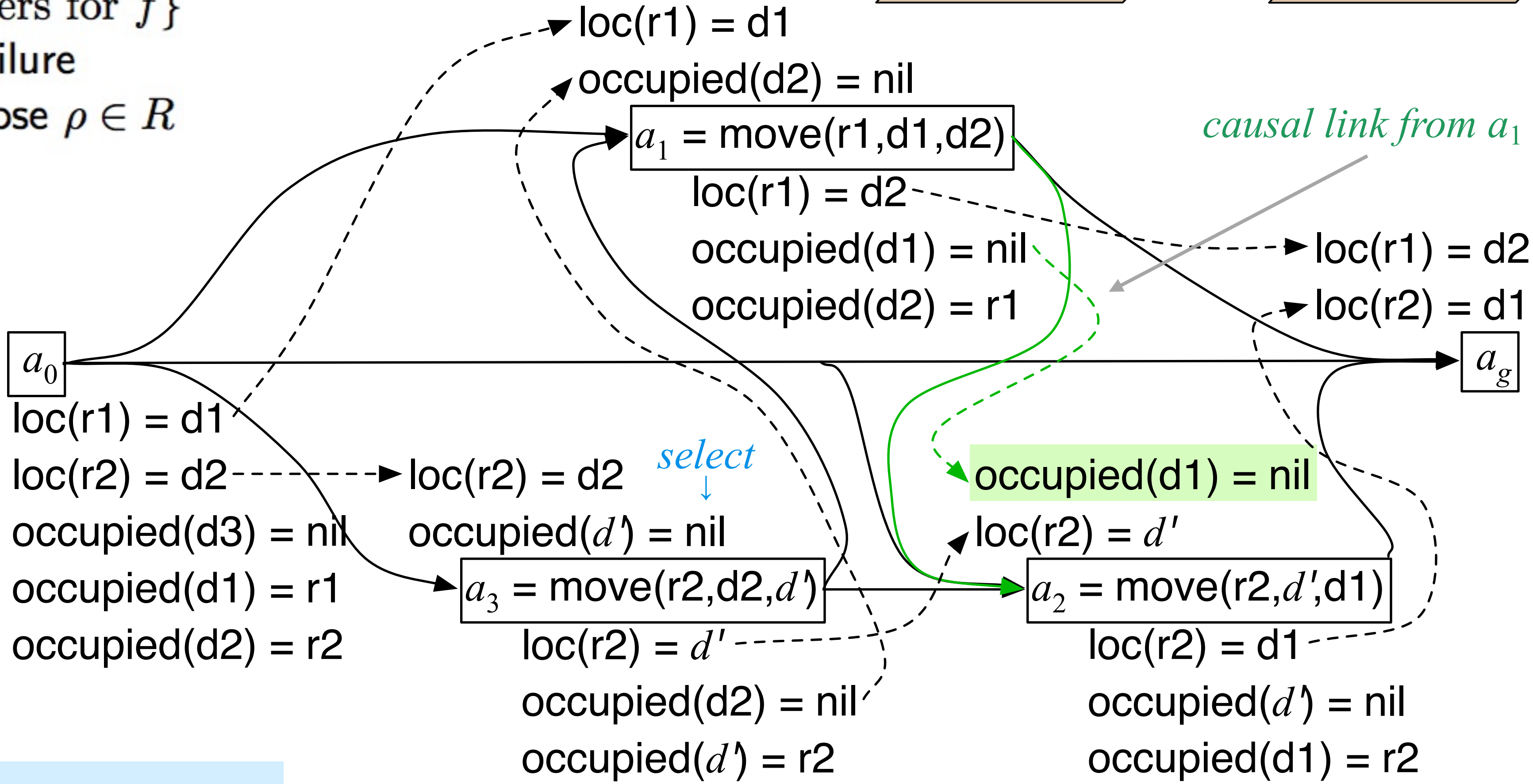
move(r, d, d')
pre: loc(r) = d, occupied(d') = nil
eff: loc(r) $\leftarrow d'$, occupied(d') = r, occupied(d) = nil

PSP Algorithm

PSP(Σ, π)
  loop
    if $Flaws(\pi) = \varnothing$ then return π
    arbitrarily select $f \in Flaws(\pi)$
    $R \leftarrow$ {all feasible resolvers for f}
    if $R = \varnothing$ then return failure
    nondeterministically choose $\rho \in R$
    $\pi \leftarrow \rho(\pi)$
  return π

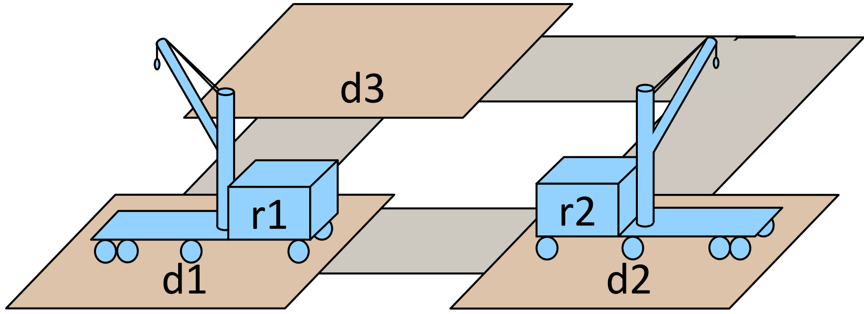


- 1 open goal
- no threats



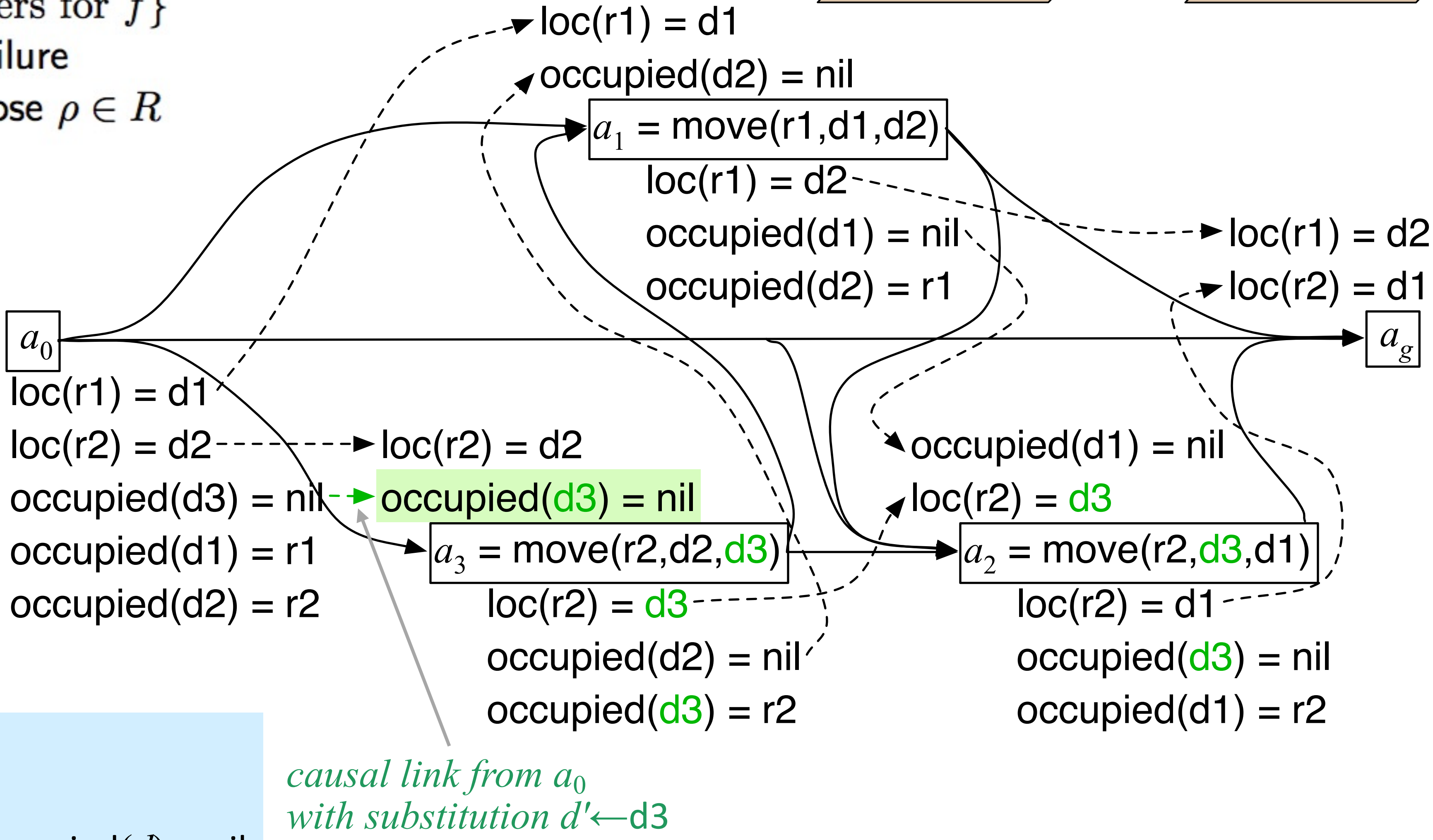
move(r, d, d')
pre: loc(r) = d, occupied(d') = nil
eff: loc(r) $\leftarrow$ d', occupied(d') = r, occupied(d) = nil

PSP Algorithm



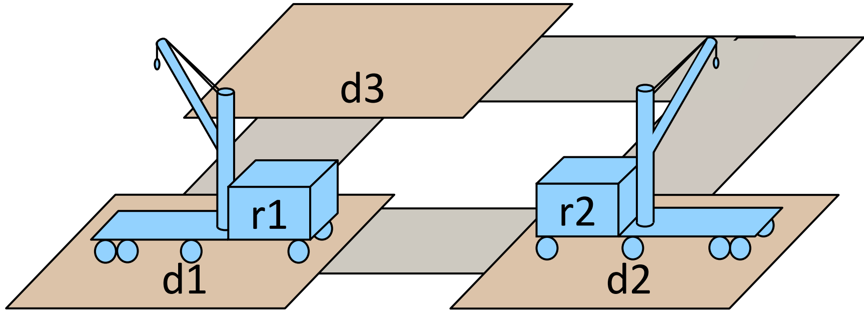
PSP(Σ, π)
    loop
        if $Flaws(\pi) = \varnothing$ then return π
        arbitrarily select $f \in Flaws(\pi)$
        $R \leftarrow$ {all feasible resolvers for f}
        if $R = \varnothing$ then return failure
        nondeterministically choose $\rho \in R$
        $\pi \leftarrow \rho(\pi)$
    return π

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



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

PSP Algorithm



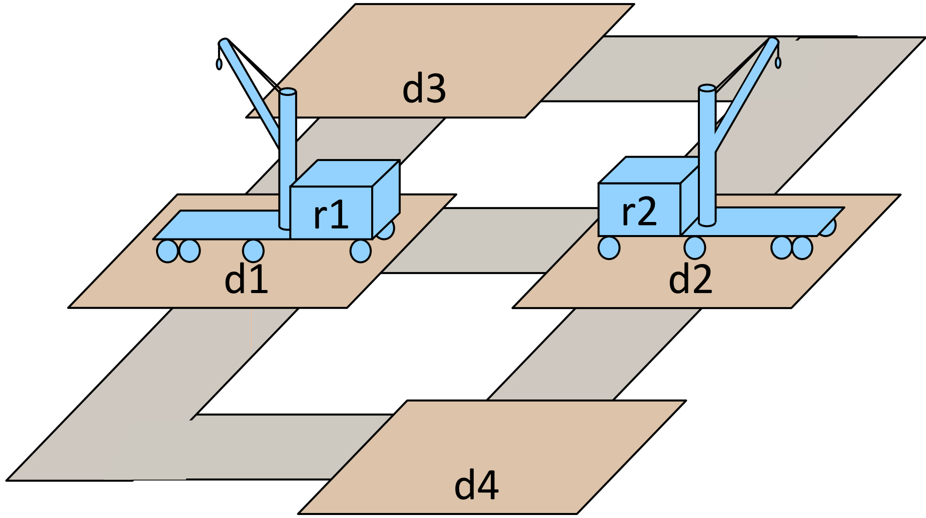
PSP(Σ, π)
  loop
    if $Flaws(\pi) = \varnothing$ then return π
    arbitrarily select $f \in Flaws(\pi)$
    $R \leftarrow$ {all feasible resolvers for f}
    if $R = \varnothing$ then return failure
    nondeterministically choose $\rho \in R$
    $\pi \leftarrow \rho(\pi)$
  return π

- The solution we found: a_0 → move(r2,d2,d3) → move(r1,d1,d2) → move(r2,d3,d1) → a_g
- Another: a_0 → move(r2,d2,d3) → move(r1,d1,d2) → move(r2,d3,d1) → move(r1,d2,d3) → move(r2,d1,d2) → move(r1,d3,d1) → move(r2,d2,d3) → move(r1,d1,d2) → move(r2,d3,d1) → a_g
- Infinitely many others

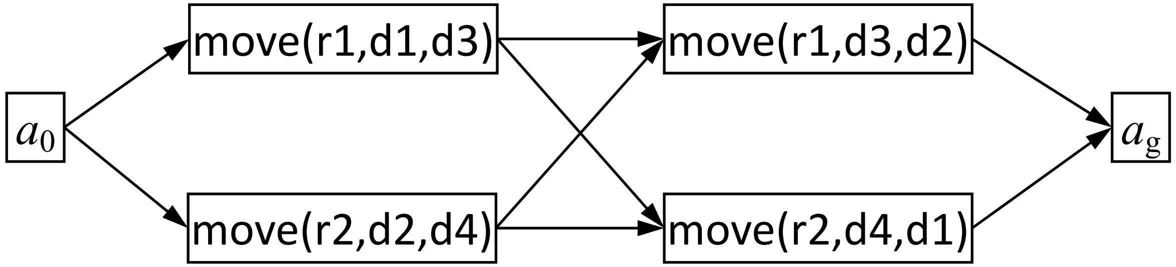
move(r, d, d')
pre: loc(r) = d, occupied(d') = nil
eff: loc(r) $\leftarrow d'$, occupied(d') = r, occupied(d) = nil

PSP Algorithm

PSP(Σ, π)
  loop
    if $Flaws(\pi) = \varnothing$ then return π
    arbitrarily select $f \in Flaws(\pi)$
    $R \leftarrow$ {all feasible resolvers for f}
    if $R = \varnothing$ then return failure
    nondeterministically choose $\rho \in R$
    $\pi \leftarrow \rho(\pi)$
  return π



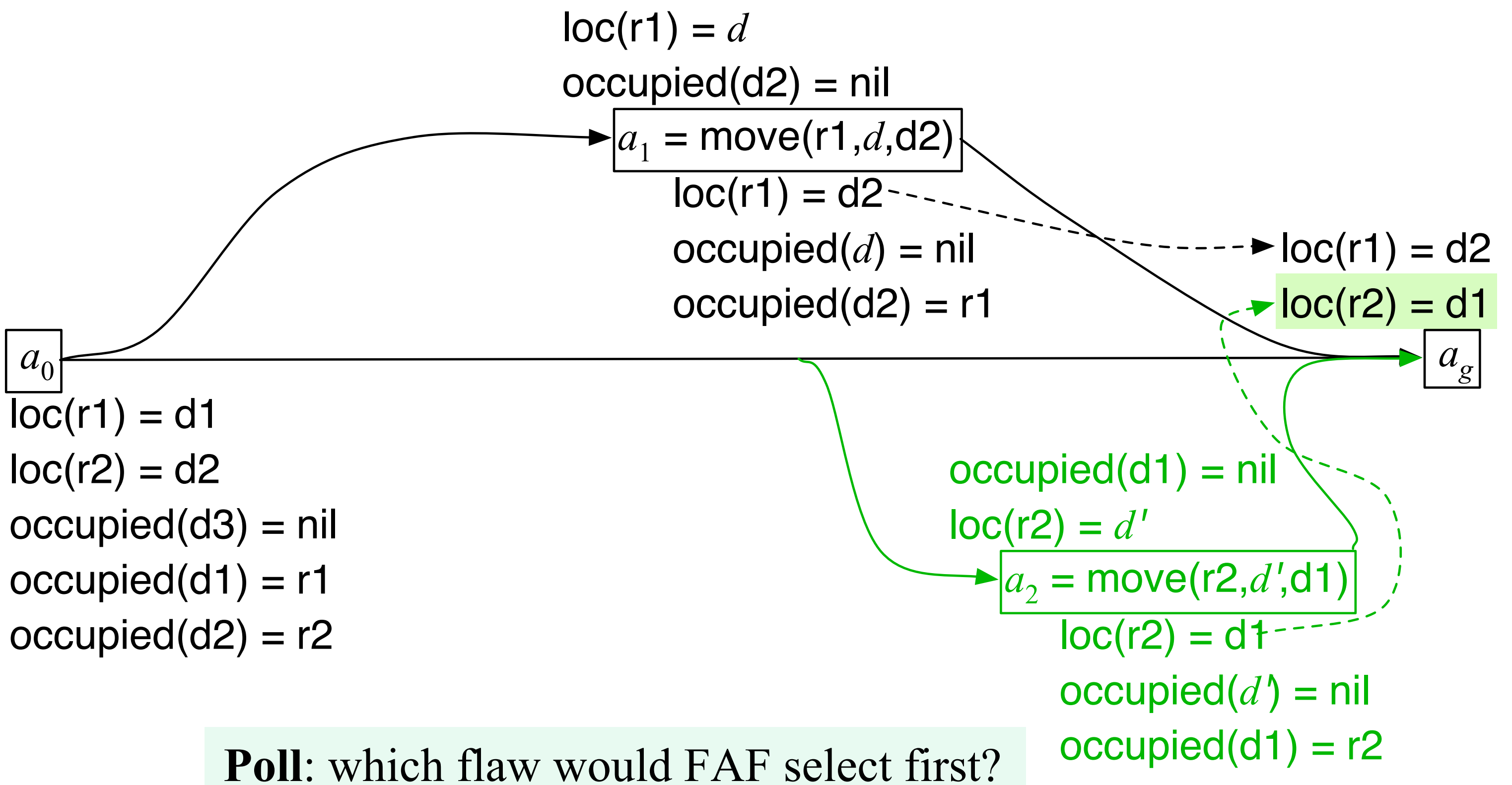
- Add another location to the initial state
- Still have all of the solutions on the previous page
- This time, PSP also can return partially-ordered solutions



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

Selecting a Flaw

PSP(Σ, π)
  loop
    if $Flaws(\pi) = \varnothing$ then return π
    arbitrarily select $f \in Flaws(\pi)$
    $R \leftarrow$ {all feasible resolvers for f}
    if $R = \varnothing$ then return failure
    nondeterministically choose $\rho \in R$
    $\pi \leftarrow \rho(\pi)$
  return π



- Selecting a flaw to resolve in PSP ≈ selecting a variable to instantiate in a CSP
  - AND-branch in both cases
- *Fewest Alternatives First (FAF):*
  - select flaw with fewest resolvers

    ≈ Minimum Remaining Values (MRV) heuristic for CSPs

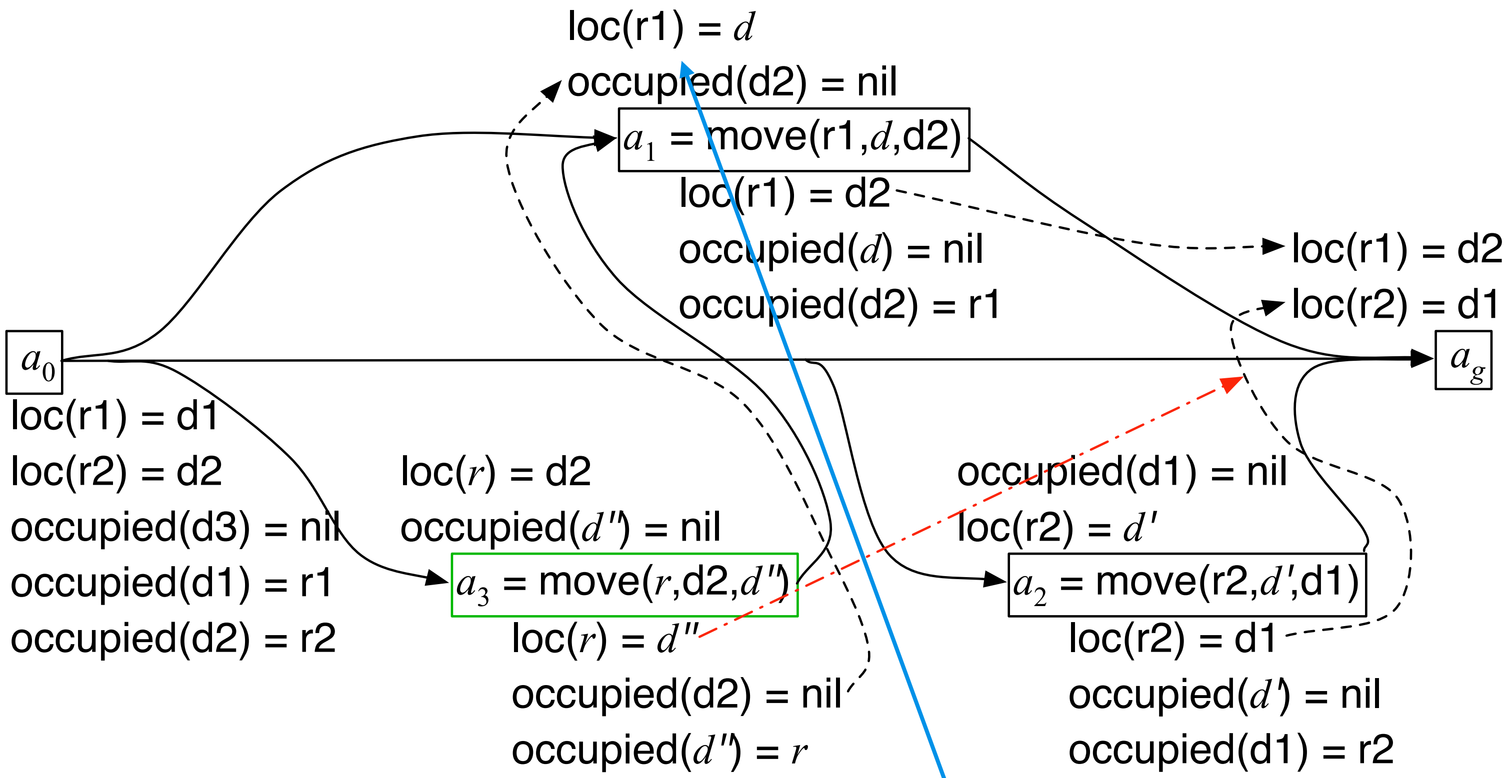
Poll: which flaw would FAF select first?

A. loc(r1)=d
B. occupied(d2) = nil
C. occupied(d2) = nil
D. loc(r2)=d′
E. no preference

move(r, d, d')
pre: loc(r) = d, occupied(d') = nil
eff: loc(r) ← d', occupied(d') = r, occupied(d) = nil

Choosing a Resolver

PSP(Σ, π)
  loop
    if $Flaws(\pi) = \varnothing$ then return π
    arbitrarily select $f \in Flaws(\pi)$
    $R \leftarrow$ {all feasible resolvers for f}
    if $R = \varnothing$ then return failure
    nondeterministically choose $\rho \in R$
    $\pi \leftarrow \rho(\pi)$
  return π



- Choosing a resolver for a flaw ≈ 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

    ≈ Least Constraining Value (LCV) heuristic for CSPs

Poll: for loc(r1)=d in a_1, which resolver would LCR choose first?

A. causal link from a new action
B. causal link from a_0, with substitution d←d1
C. causal link from a_3, with substitutions r←r1, d''←d
D. no preference

move(r, d, d')
pre: loc(r) = d, occupied(d') = nil
eff: loc(r) ← d', occupied(d') = r, occupied(d) = nil

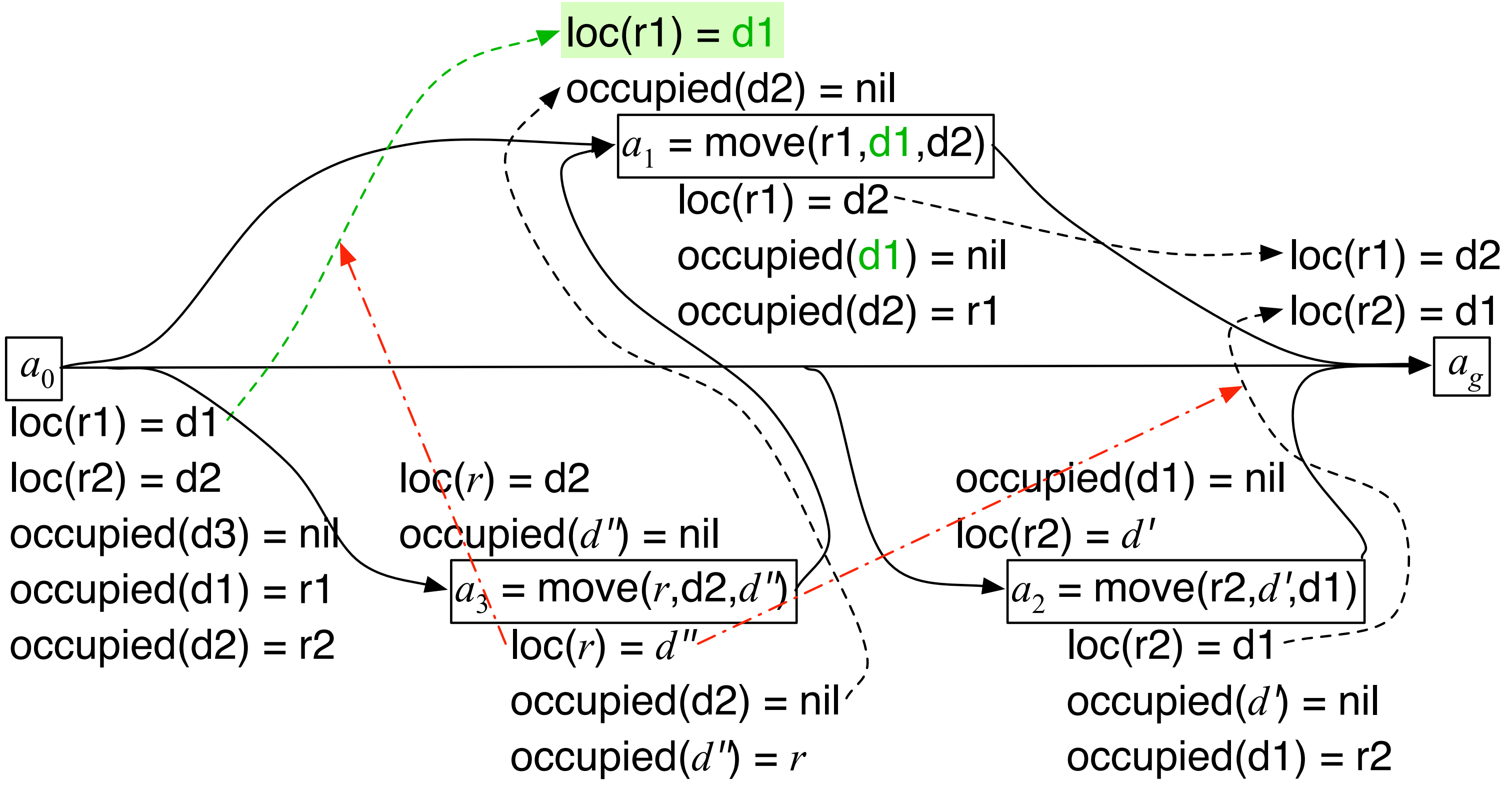
Choosing a Resolver

PSP(Σ, π)
  loop
    if $Flaws(\pi) = \varnothing$ then return π
    arbitrarily select $f \in Flaws(\pi)$
    $R \leftarrow$ {all feasible resolvers for f}
    if $R = \varnothing$ then return failure
    nondeterministically choose $\rho \in R$
    $\pi \leftarrow \rho(\pi)$
  return π

- ~~*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

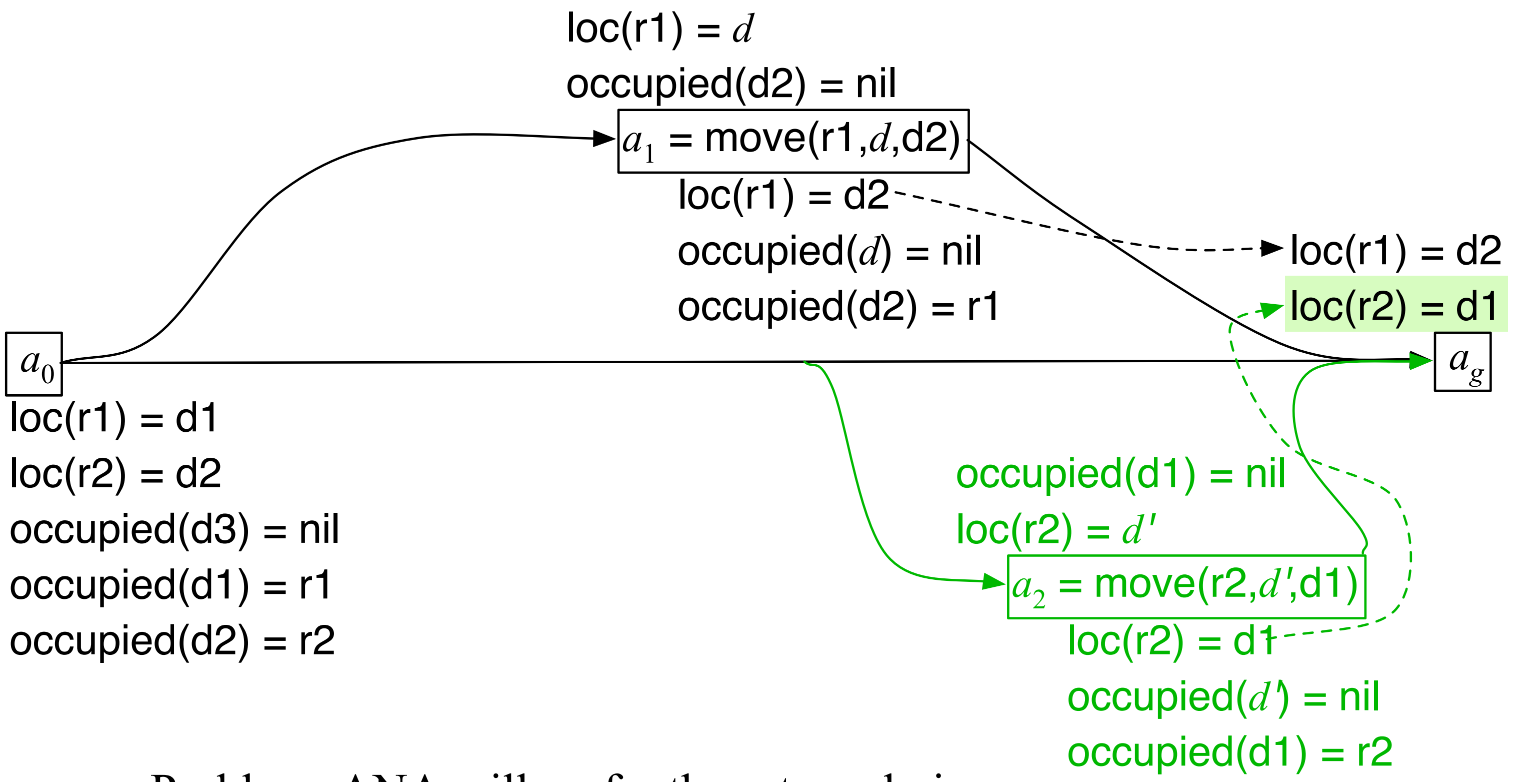
Perhaps this might work:

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



PSP(Σ, π)
  loop
    if $Flaws(\pi) = \varnothing$ then return π
    arbitrarily select $f \in Flaws(\pi)$
    $R \leftarrow$ {all feasible resolvers for f}
    if $R = \varnothing$ then return failure
    nondeterministically choose $\rho \in R$
    $\pi \leftarrow \rho(\pi)$
  return π

Choosing a Resolver



- ~~*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

- 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 ⇒ need to backtrack
- Perhaps use ANA anyway?

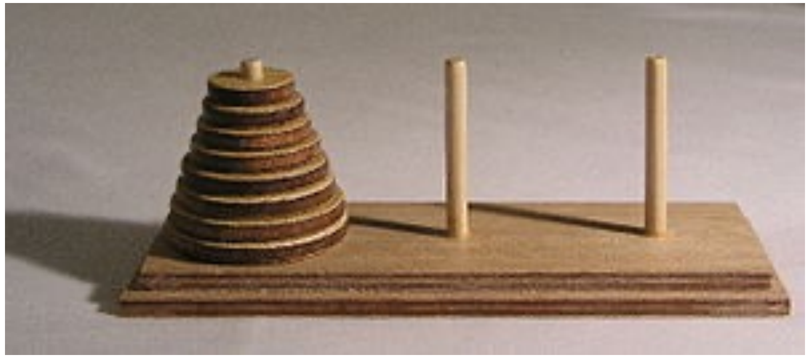
Discussion

- 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

$$\cdots \longrightarrow s \longrightarrow s' \longrightarrow s$$

- Prune if π contains the same *action* more than once?

  ⟨a1, a2, …, a1, …⟩
  - 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

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