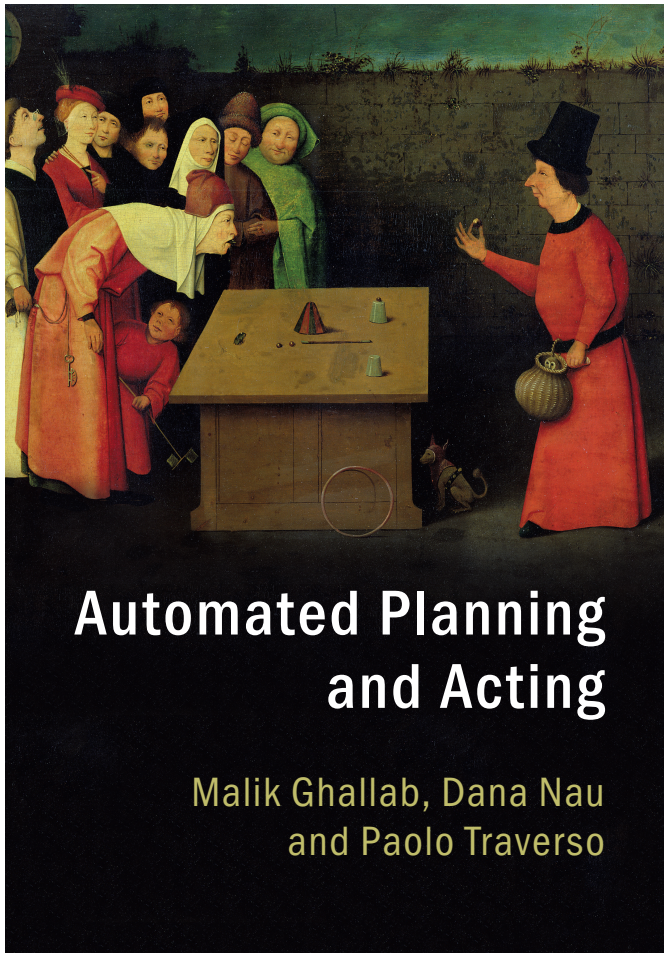
Chapter 2

Deliberation with Deterministic Models

2.3: Heuristic Functions
2.4: Backward Search
2.5: Plan-Space Search

Adapted from Dana S. Nau
University of Maryland



http://www.laas.fr/planning

Motivation

- Given: planning problem P in domain Σ
- One way to create a heuristic function:
  - Weaken some of the constraints, get additional solutions
  - *Relaxed* planning domain Σ' and relaxed problem $P' = (\Sigma', s_0, g')$ such that
    - every solution for P is also a solution for P'
    - additional solutions with lower cost
  - Suppose we have an algorithm A for solving planning problems in Σ'
    - Heuristic function $h_A(s)$ for P:
      - Find a solution π' for (Σ', s, g'); return cost(π')
      - Useful if A runs quickly
    - If A always finds optimal solutions, then h_A is admissible

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

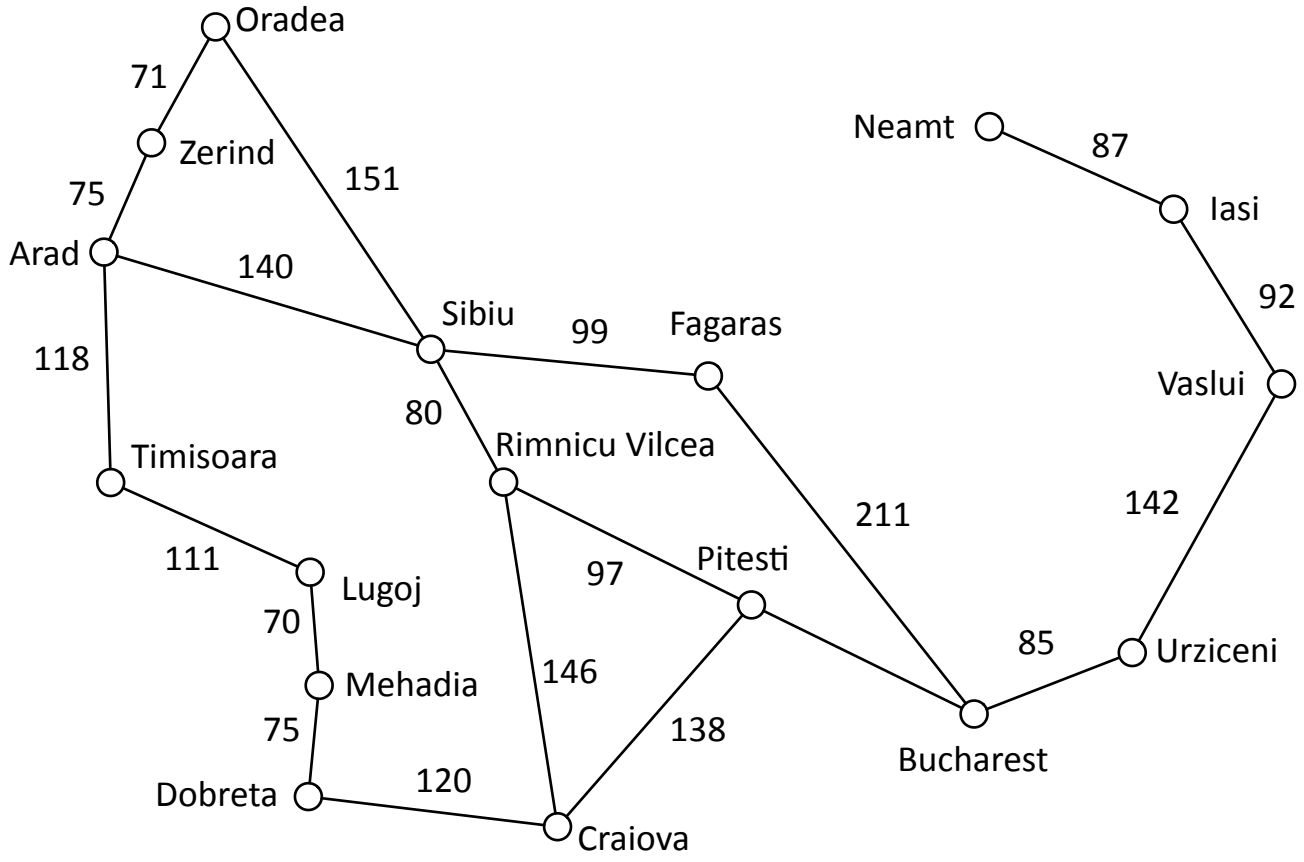
Chapter 2, part *b* (chap2b.pdf):

Next ⟶	2.3	Heuristic functions
	2.4	Backward search
	2.5	Plan-space search

Example

- Relaxation: let vehicle travel in a straight line between any pair of cities
  - straight-line-distance ≤ distance by road

    ⇒ additional solutions with lower cost



straight-line dist.
from s to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Fagaras	176
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Domain-independent Heuristics

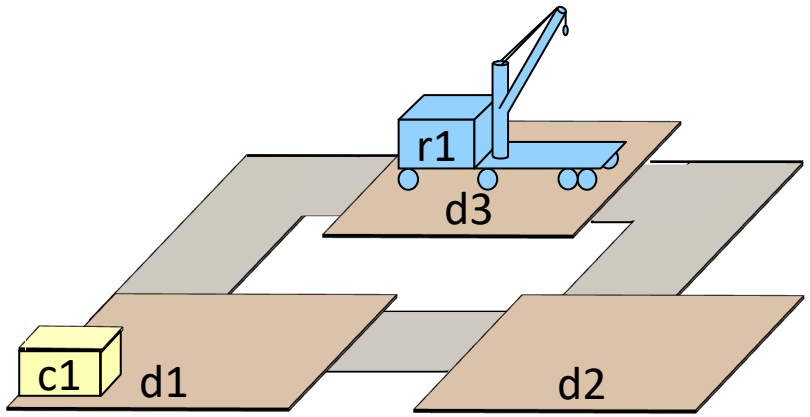
- Use relaxation to get heuristic functions that can be used in *any* classical planning problem
  - Additive-cost heuristic
  - Max-cost heuristic

  } In the book, but I'll skip them
  - Delete-relaxation heuristics
    - Optimal relaxed solution
    - Fast-forward heuristic
  - Landmark heuristics

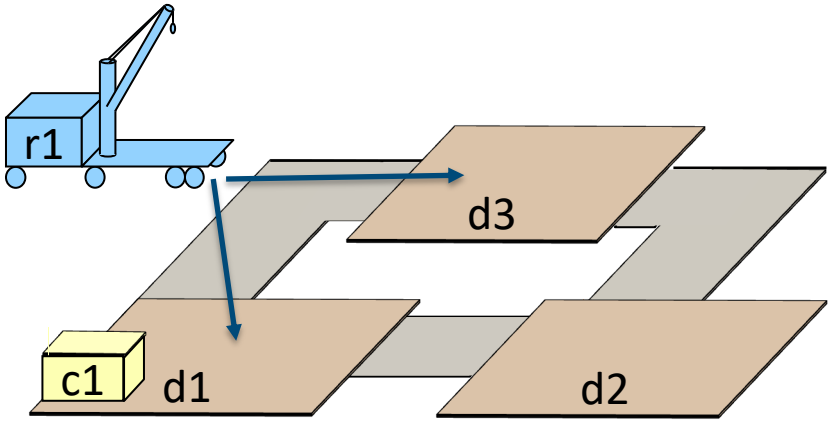
2.3.2 Delete-Relaxation

- Allow a state variable to have more than one value at the same time
- When assigning a new value, keep the old one too
- *Relaxed state-transition function*, γ^+
  - If action a is applicable to state s, then $\gamma^+(s,a) = s \cup \gamma(s,a)$

- If s includes an atom $x=v$, and a has an effect $x \leftarrow w$
  - Then $\gamma^+(s,a)$ includes both $x=v$ and $x=w$
- *Relaxed state* (or *r-state*)
  - a set $\hat{s}$ of ground atoms that includes ≥ 1 value for each state variable
  - represents {all states that are subsets of $\hat{s}$}



move(r1, d3, d1)
pre: loc(r1) = d3
eff: **loc(r1) ← d1**



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

$\hat{s}_1 = \gamma^+(s_0, \text{move(r1,d3,d1)})$
= {**loc(r1)=d3, loc(r1)=d1,** cargo(r1)=nil, loc(c1)=d1}

Poll: would the following definition be equivalent?

- Action a is *r-applicable* in $\hat{s}$ if $\hat{s}$ satisfies a's preconditions

Relaxed Applicability

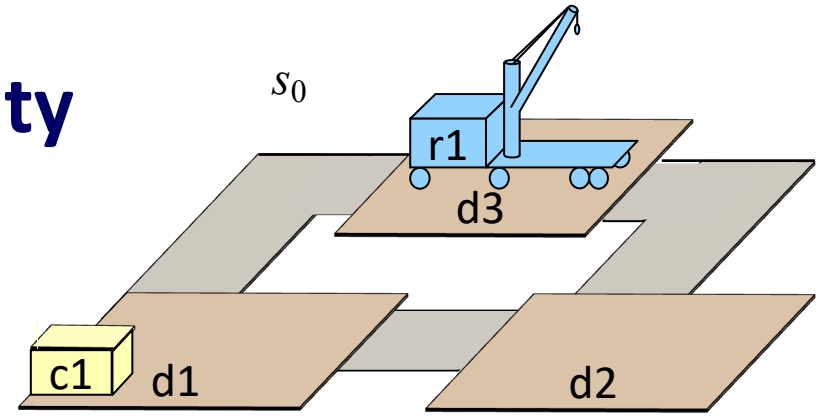
- Action a is *r-applicable* in a relaxed state $\hat{s}$ if a subset of $\hat{s}$ satisfies a's preconditions
- If a is r-applicable then $\gamma^+(\hat{s},a) = \hat{s} \cup \gamma(s,a)$

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

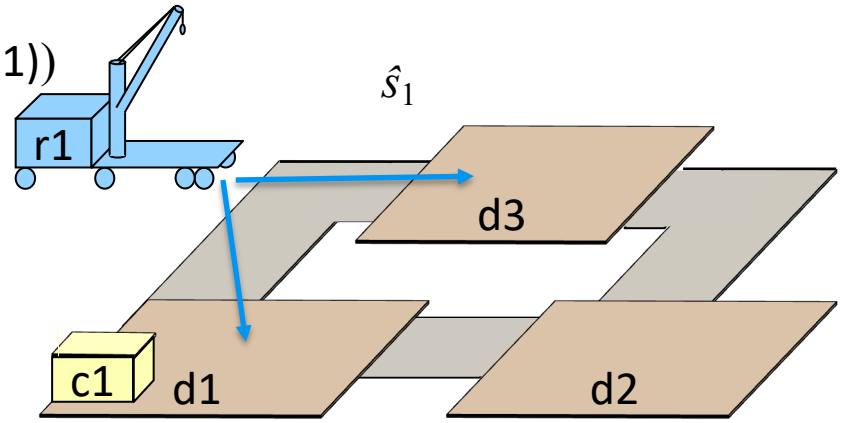
move(r, d, e)
  pre: loc(r)=d
  eff: loc(r)←e

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

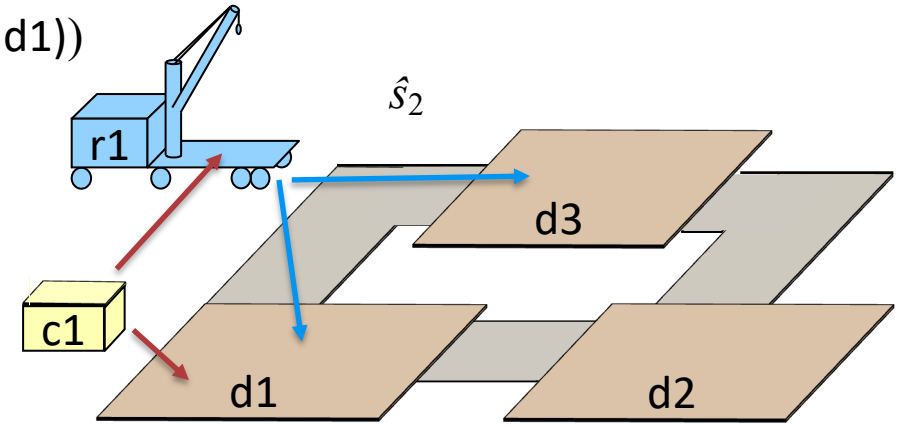
s_0 = {loc(r1) = d3,
  cargo(r1) = nil,
  loc(c1) = d1}



$\hat{s}_1 = \gamma^+(s_0,$ move(r1,d3,d1))
  = {loc(r1) = d1,
  loc(r1) = d3,
  cargo(r1) = nil,
  loc(c1) = d1}



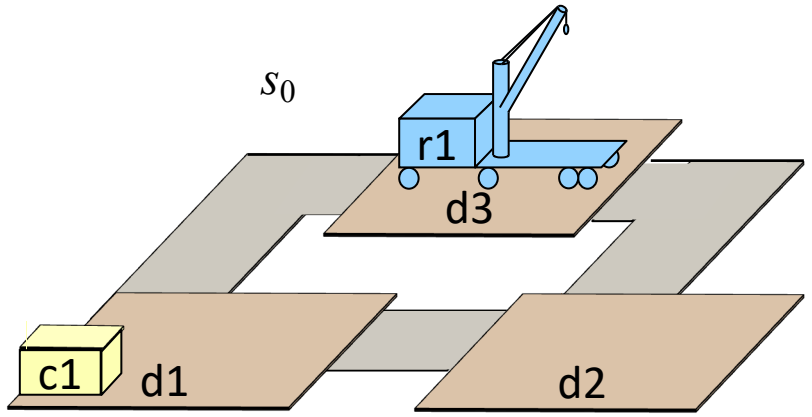
$\hat{s}_2 = \gamma^+(\hat{s}_1,$ load(r1,c1,d1))
  = {loc(r1) = d1,
  loc(r1) = d3,
  cargo(r1) = nil,
  cargo(r1) = c1,
  loc(c1) = r1,
  loc(c1) = d1}



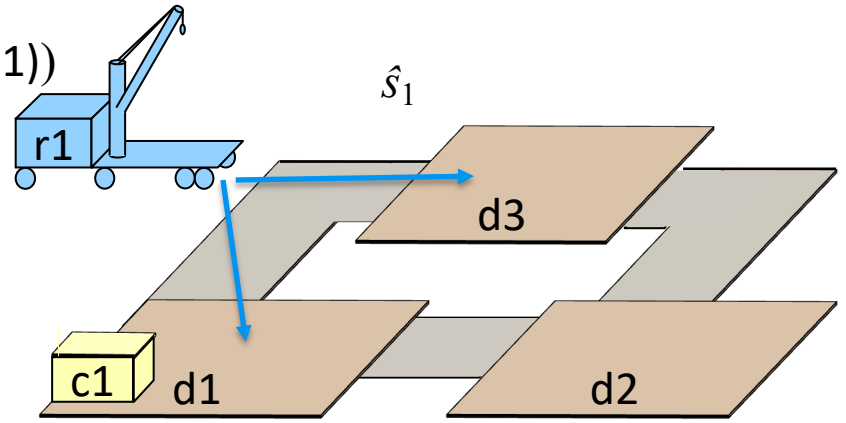
Relaxed Applicability (continued)

- Let $\pi = \langle a_1, \ldots, a_n \rangle$ be a plan
- Suppose we can r-apply the actions of π in the order $a_1, \ldots, a_n$:
  - r-apply a_1 in $\hat{s}_0$, get $\hat{s}_1 = \gamma^+(\hat{s}_0, a_1)$
  - r-apply a_2 in $\hat{s}_1$, get $\hat{s}_2 = \gamma^+(\hat{s}_1, a_2)$
  - …
  - r-apply a_n in $\hat{s}_{n-1}$, get $\hat{s}_n = \gamma^+(\hat{s}_{n-1}, a_n)$
- Then π is *r-applicable* in $\hat{s}_0$ and $\gamma^+(\hat{s}_0, \pi) = \hat{s}_n$
- Example: if s_0 and $\hat{s}_2$ are as shown, then $\gamma^+(s_0, \langle \text{move(r1,d3,d1), load(r1,c1,d1)} \rangle) = \hat{s}_2$

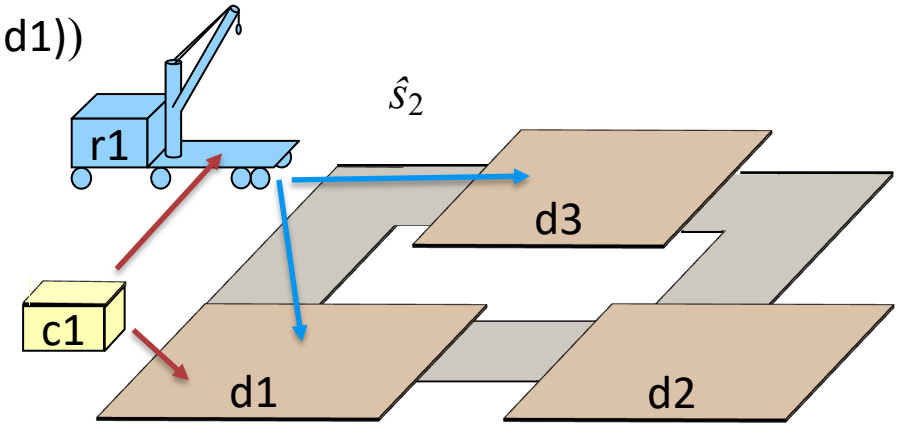
s_0 = {loc(r1) = d3,
cargo(r1) = nil,
loc(c1) = d1}



$\hat{s}_1 = \gamma^+(s_0, \text{move(r1,d3,d1)})$
= {loc(r1) = d1,
loc(r1) = d3,
cargo(r1) = nil,
loc(c1) = d1}



$\hat{s}_2 = \gamma^+(\hat{s}_1, \text{load(r1,c1,d1)})$
= {loc(r1) = d1,
loc(r1) = d3,
cargo(r1) = nil,
cargo(r1) = c1,
loc(c1) = r1,
loc(c1) = d1}



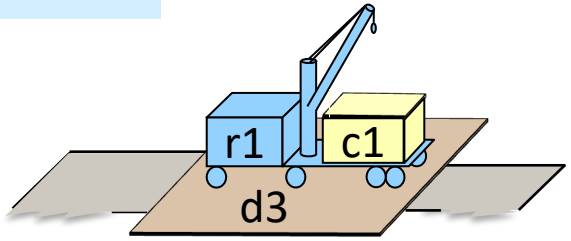
Relaxed Solution

- An r-state $\hat{s}$ *r-satisfies* a formula g if a subset of $\hat{s}$ satisfies g
- *Relaxed solution* for a planning problem $P = (\Sigma, s_0, g)$:
  - a plan π such that $\gamma^+(s_0, \pi)$ r-satisfies g
- Example: let (Σ, s_0, g) be as shown
  - ⟨move(r1,d3,d1), load(r1,c1,d1)⟩ is a relaxed solution

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

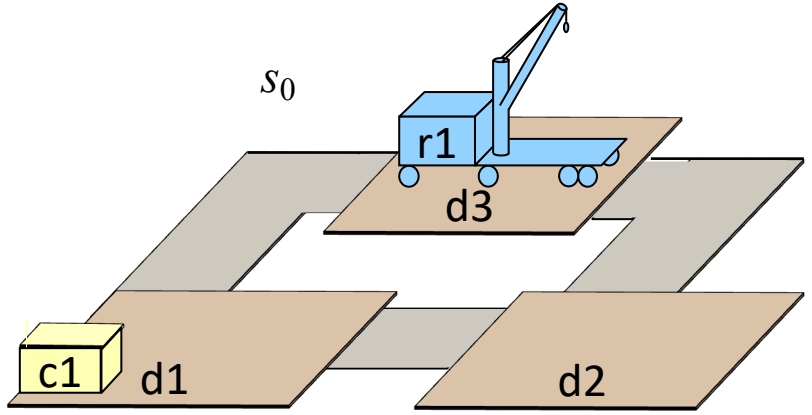
move(r, d, e)
  pre: loc(r)=d
  eff: loc(r)←e

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

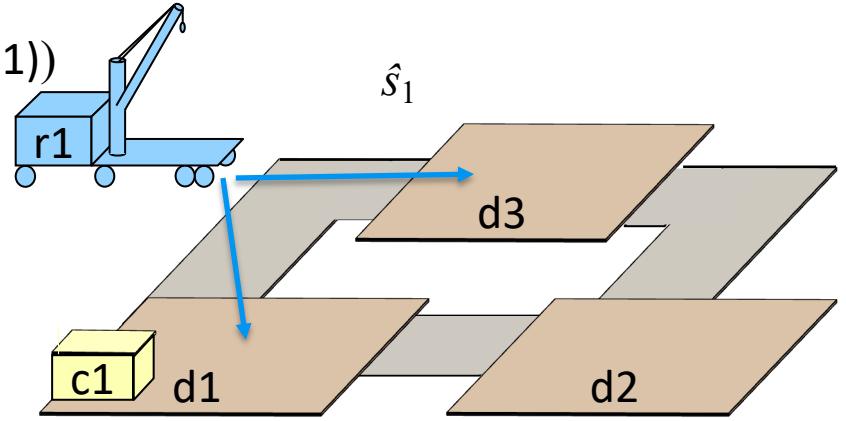


g = {loc(r1)=d3, loc(c1)=r1}

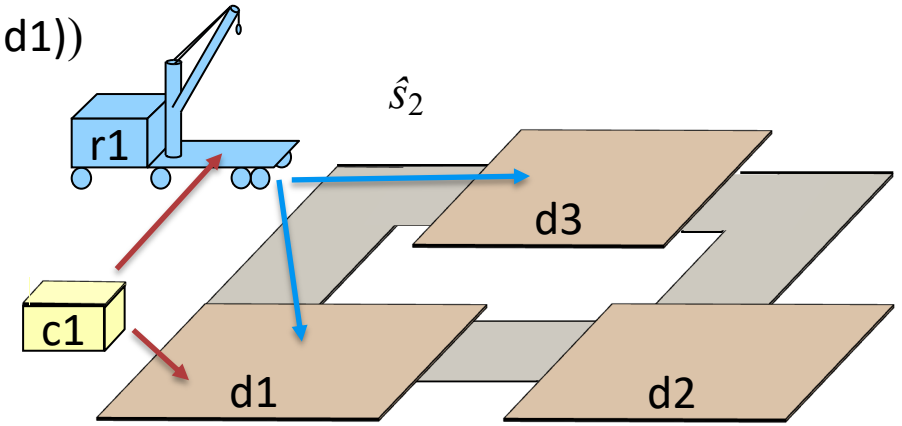
s_0 = {loc(r1) = d3,
cargo(r1) = nil,
loc(c1) = d1}



$\hat{s}_1 = \gamma^+(s_0,$ move(r1,d3,d1))
= {loc(r1) = d1,
loc(r1) = d3,
cargo(r1) = nil,
loc(c1) = d1}



$\hat{s}_2 = \gamma^+(\hat{s}_1,$ load(r1,c1,d1))
= {loc(r1) = d1,
loc(r1) = d3,
cargo(r1) = nil,
cargo(r1) = c1,
loc(c1) = r1,
loc(c1) = d1}



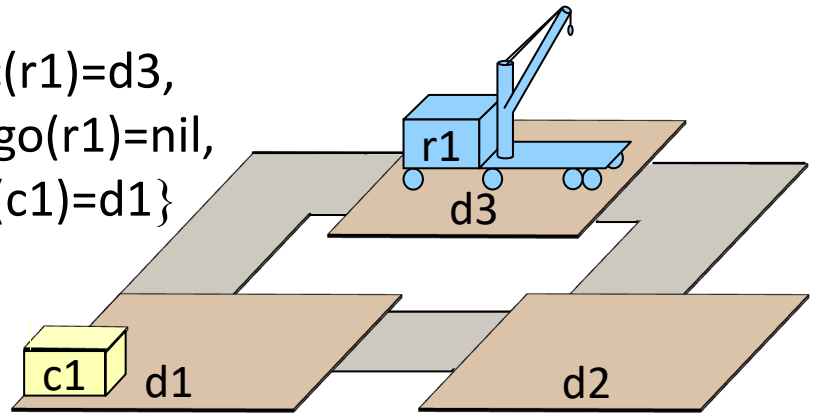
Optimal Relaxed Solution Heuristic

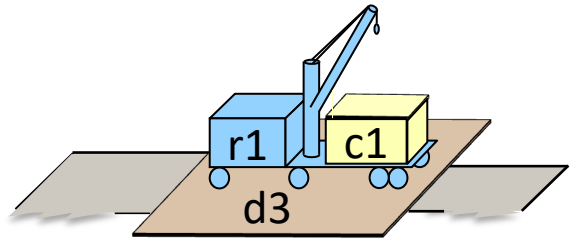
- Planning problem $P = (\Sigma, s_0, g)$
- *Optimal relaxed solution* heuristic:
  - $h^+(s)$ = minimum cost of all relaxed solutions for (Σ, s, g)

```
move(r1, d3, d1)
    pre: loc(r1) = d3
    eff: loc(r1) ← d1

load(r1,c1,d1)
    pre: cargo(r1)=nil,
         loc(c1)=d1,
         loc(r1)=d1
    eff: cargo(r1) ← c1,
         loc(c1) ← r1
```

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





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

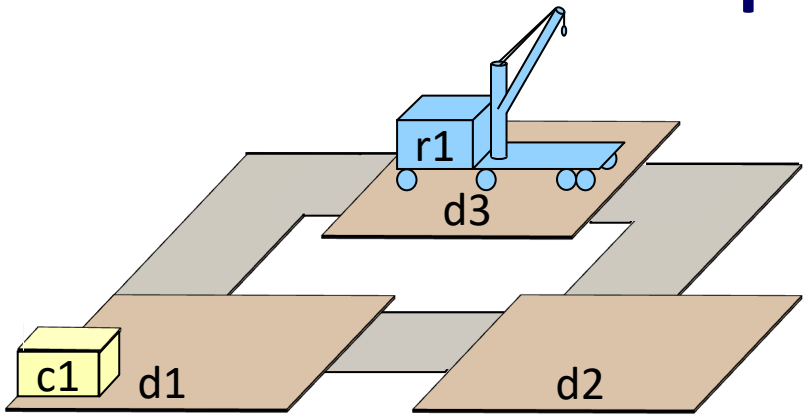
- Relaxed solution π = ⟨move(r1,d3,d1), load(r1,c1,d1)⟩
  - cost(π) = 2
- No less-costly relaxed solution, so $h^+(s_0) = 2$

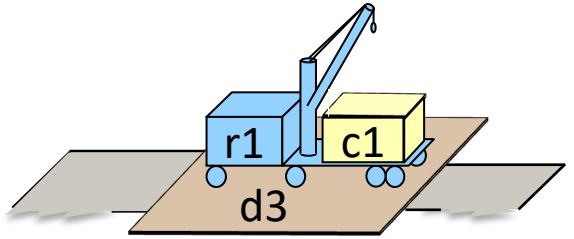
Poll: is h^+ admissible?

1. Yes
2. No

Example: GBFS

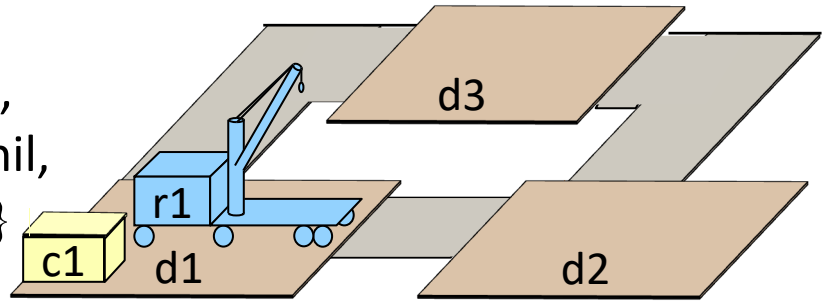
s_0 = {loc(r1)=d3,
cargo(r1)=nil,
loc(c1)=d1}



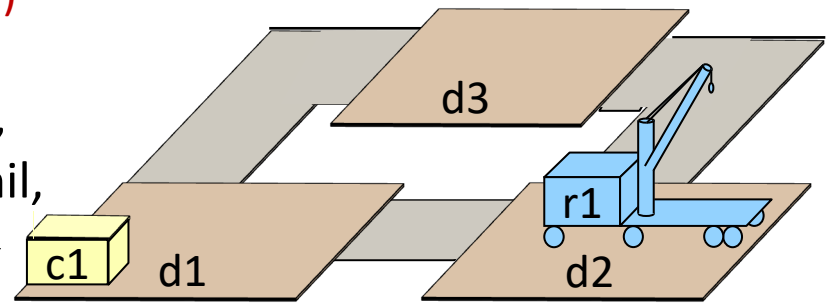


g = {loc(r1)=d3, loc(c1)=r1}

a_1 = move(r1,d3,d1)
$s_1 = \gamma(s_0,a_1)$
= {loc(r1) = d1,
cargo(r1) = nil,
loc(c1) = d1}



a_2 = move(r1,d3,d2)
$s_2 = \gamma(s_0,a_2)$
= {loc(r1) = d2,
cargo(r1) = nil,
loc(c1) = d1}



- GBFS with initial state s_0, goal g, heuristic h^+
- Two applicable actions: a_1, a_2
- Resulting states: s_1, s_2
- GBFS computes $h^+(s_1)$ and $h^+(s_2)$
  - Chooses the state that has the lower h^+ value

Poll 1: What is $h^+(s_1)$?

1. 1
2. 2
3. 3
4. 4
5. other

Poll 2: What is $h^+(s_2)$?

1. 1
2. 2
3. 3
4. 4
5. other

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

move(r, d, e)
pre: loc(r)=d
eff: loc(r)←e

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

Fast-Forward Heuristic

- Every state is also a relaxed state
- Every solution is also a relaxed solution

- $h^+(s)$ = minimum cost of all relaxed solutions
  - Thus h^+ is admissible
  - Problem: computing it is NP-hard

- Fast-Forward Heuristic, h^{FF}
  - An approximation of h^+ that's easier to compute
    - Upper bound on h^+
  - Name comes from a planner called *Fast Forward*

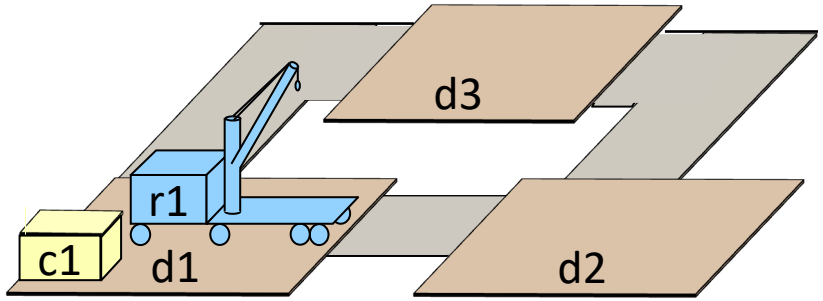
Preliminaries

- Suppose a_1 and a_2 are r-applicable in $\hat{s}_0$
- Let $\hat{s}_1 = \gamma^+(\hat{s}_0, a_1) = \hat{s}_0 \cup \text{eff}(a_1)$
- Then a_2 is still applicable in $\hat{s}_1$
  - $\hat{s}_2 = \gamma^+(\hat{s}_1, a_2) = \hat{s}_0 \cup \text{eff}(a_1) \cup \text{eff}(a_2)$
- Apply a_1 and a_2 in the opposite order ⇒ same state $\hat{s}_2$

- Let A_1 be a set of actions that all are r-applicable in s_0
  - Can r-apply them in any order and get same result
  - $\hat{s}_1 = \gamma^+(\hat{s}_0, A_1) = \hat{s}_0 \cup \text{eff}(A_1)$
    - where $\text{eff}(A) = \bigcup\{\text{eff}(a) \mid a \in A\}$
- Suppose A_2 is a set of actions that are r-applicable in $\hat{s}_1$
  - $\hat{s}_2 = \gamma^+(\hat{s}_0, \langle A_1, A_2 \rangle) = \hat{s}_0 \cup \text{eff}(A_1) \cup \text{eff}(A_2)$

…

- Define $\gamma^+(\hat{s}_0, \langle A_1, A_2, \ldots, A_n \rangle)$ in the obvious way



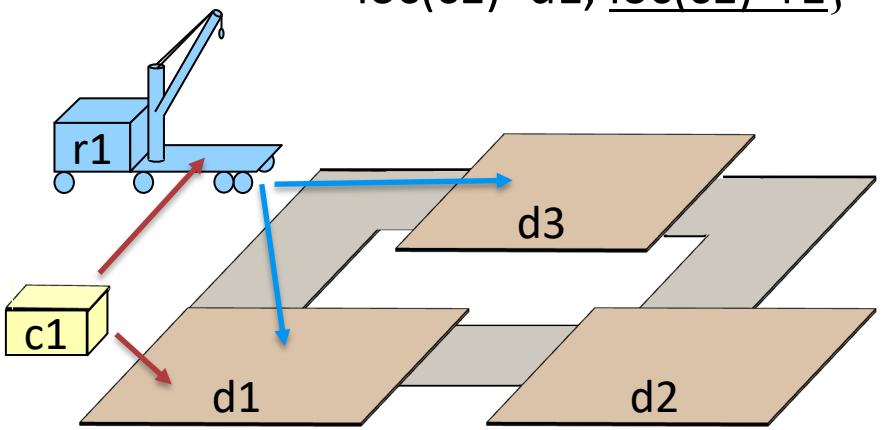
s_0 = {loc(r1)=d1, cargo(r1)=nil, loc(c1)=d1}

a_1 = load(r1,c1,d1)

a_2 = move(r1,d1,d3)

$A_1 = \{a_1, a_2\}$

$\gamma^+(s_0, A_1)$ = {loc(r1)=d1, loc(r1)=d3,
cargo(r1)=nil, cargo(r1)=c1,
loc(c1)=d1, loc(c1)=r1}



Fast-Forward Heuristic h^{FF}: algorithm HFF

i.e., no proper subset is a relaxed solution

HFF(Σ, s, g): // *find a minimal relaxed solution, return its cost*

1. At each iteration, include all r-applicable actions

// *construct a relaxed solution* $\langle A_1, A_2, \ldots, A_k \rangle$:
$\hat{s}_0 \leftarrow s$
for $k = 1$ by 1 until a subset of $\hat{s}_k$ r-satisfies g
    $A_k \leftarrow$ {all actions r-applicable in $\hat{s}_{k-1}$}; $\hat{s}_k \leftarrow \gamma^+(s_{k-1}, A_k)$
    if $k > 1$ and $\hat{s}_k = \hat{s}_{k-1}$ then return ∞ // *there's no solution*

2. At each iteration, choose a minimal set of actions that r-achieve $\hat{g}_i$

// *extract minimal relaxed solution* $\langle \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_k \rangle$:
$\hat{g}_k \leftarrow g$
for $i = k, k-1, \ldots, 1$:
    $\hat{a}_i \leftarrow$ any minimal subset of A_i such that $\gamma^+(\hat{s}_{i-1}, \hat{a}_i)$ r-satisfies $\hat{g}_i$
    $\hat{g}_{i-1} \leftarrow (\hat{g}_i \setminus \text{eff}(\hat{a}_i)) \cup \text{pre}(\hat{a}_i)$
return $\sum$ costs of the actions in $\hat{a}_1, \ldots, \hat{a}_k$ // *upper bound on* h^+

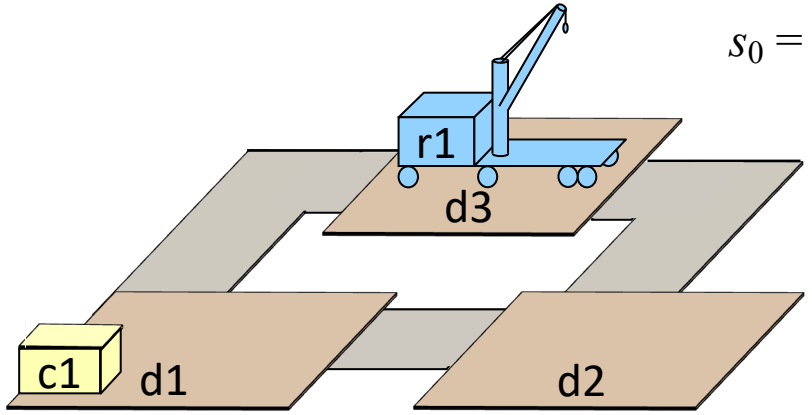
$\text{pre}(\hat{a}_i) = \bigcup \{\text{pre}(a) \mid a \in \hat{a}_i\}$
$\text{eff}(\hat{a}_i) = \bigcup \{\text{eff}(a) \mid a \in \hat{a}_i\}$

ambiguous →

- Define $h^{FF}(s)$ = the value returned by HFF(Σ, s, g)

Example: GBFS Again

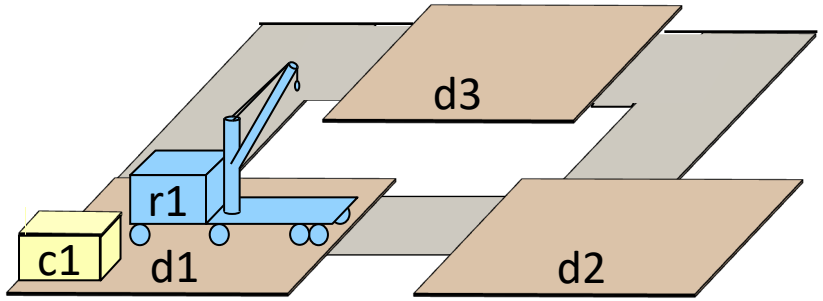
- GBFS with initial state s_0, goal g, heuristic h^{FF}
- Two applicable actions: a_1, a_2
- Resulting states: s_1, s_2
- GBFS computes $h^{FF}(s_1)$ and $h^{FF}(s_2)$
  - Chooses the state that has the lower h^{FF} value
- Next several slides:
  - $h^{FF}(s_1)$
  - $h^{FF}(s_2)$



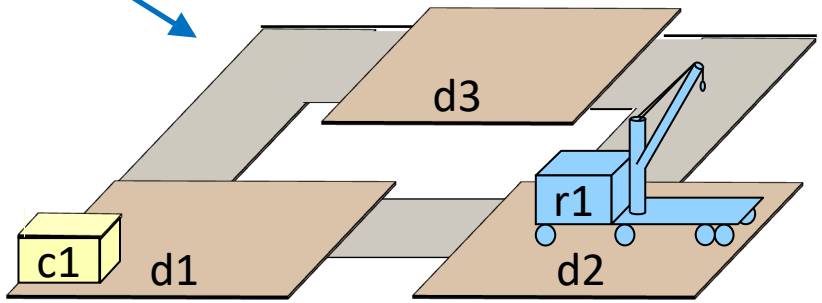
s_0 = {loc(c1) = d1, loc(r1) = d3, cargo(r1) = nil}

a_1 = move(r1,d3,d1)

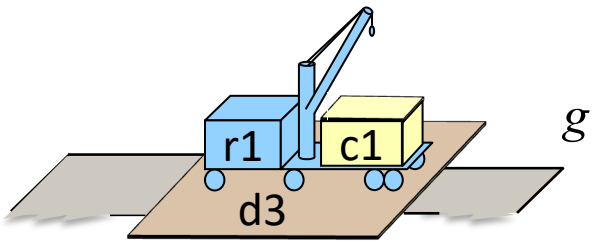
a_2 = move(r1,d3,d2)



$s_1 = \gamma(s_0,a_1)$ = {loc(c1) = d1, loc(r1) = d1, cargo(r1) = nil}



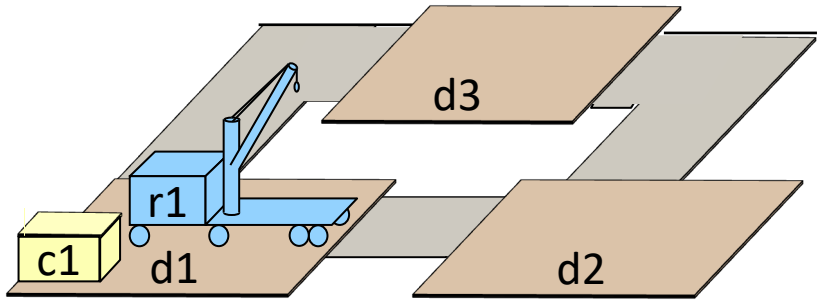
$s_2 = \gamma(s_0,a_2)$ = {loc(c1) = d1, loc(r1) = d2, cargo(r1) = nil}



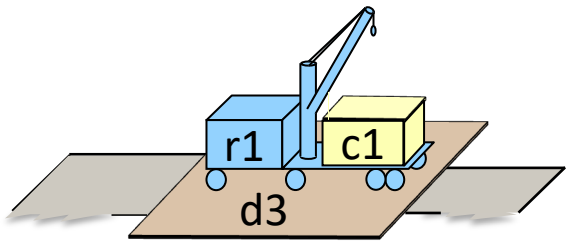
g = {loc(r1)=d3, loc(c1)=r1}

Example

- Computing $h^{FF}(s_1)$
  - 1. construct a relaxed solution
    - at each step, include all r-applicable actions



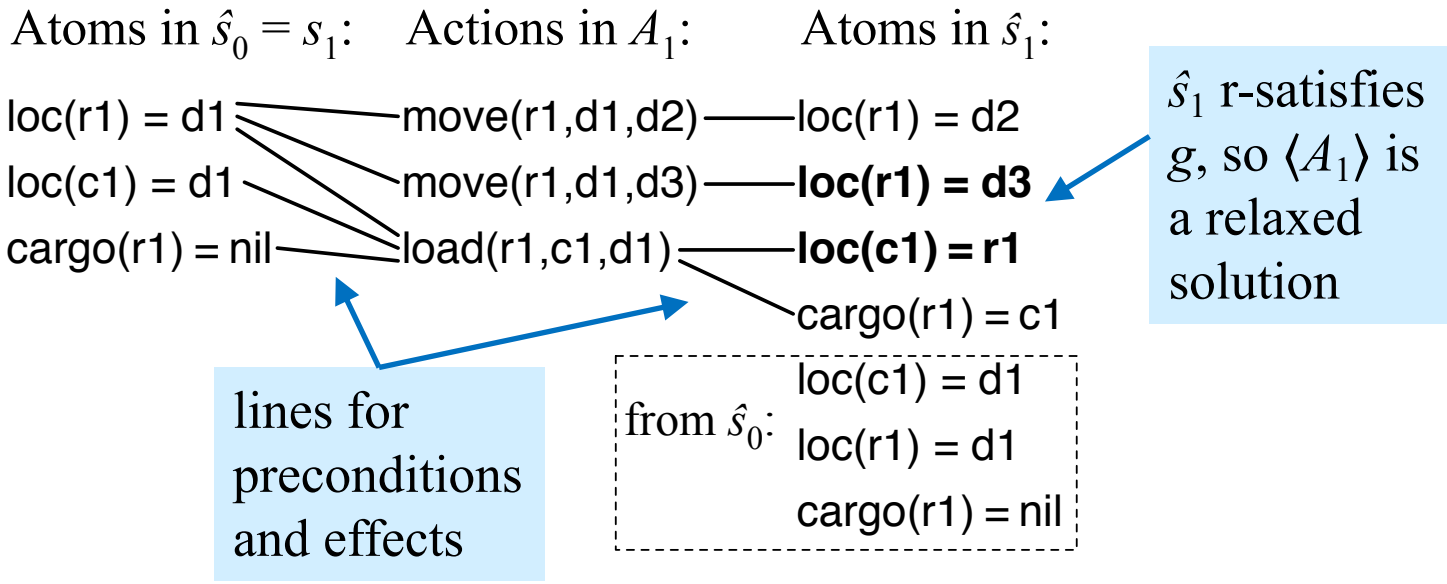
s_1 = {loc(r1)=d1, cargo(r1)=nil, loc(c1)=d1}



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

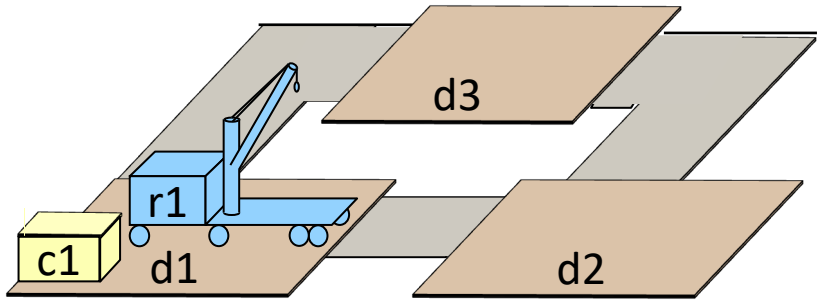
// construct a relaxed solution $\langle A_1, A_2, \ldots, A_k \rangle$:
$\hat{s}_0 \leftarrow s$
for $k = 1$ by 1 until a subset of $\hat{s}_k$ r-satisfies g
  $A_k \leftarrow$ {all actions r-applicable in $\hat{s}_{k-1}$}; $\hat{s}_k \leftarrow \gamma^+(s_{k-1}, A_k)$
  if $k > 1$ and $\hat{s}_k = \hat{s}_{k-1}$ then return ∞

Relaxed Planning Graph (RPG) starting at $\hat{s}_0 = s_1$

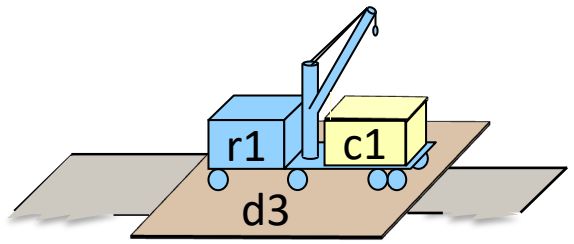


Example

- Computing $h^{FF}(s_1)$
  2. extract a *minimal* relaxed solution
     - if you remove any actions from it, it's no longer a relaxed solution



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



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

// extract minimal relaxed solution $\langle \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_k \rangle$:

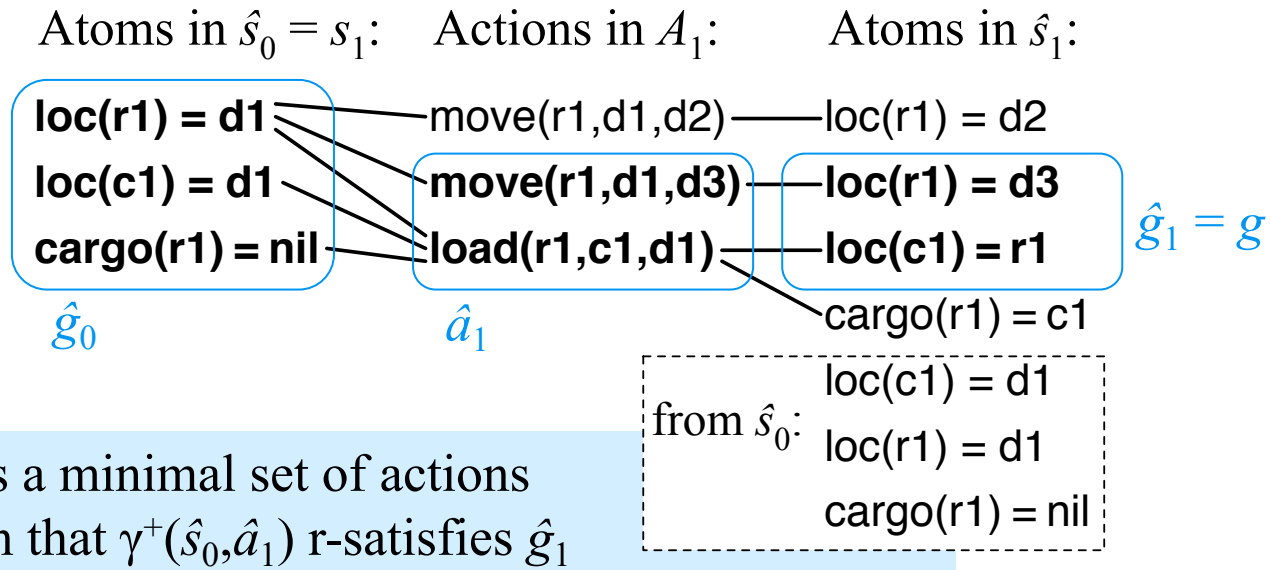
$\hat{g}_k \leftarrow g$

for $i = k, k-1, \ldots, 1$:

  $\hat{a}_i \leftarrow$ any minimal subset of A_i such that $\gamma^+(\hat{s}_{i-1},\hat{a}_i)$ r-satisfies $\hat{g}_i$

  $\hat{g}_{i-1} \leftarrow (\hat{g}_i \setminus \text{eff}(\hat{a}_i)) \cup \text{pre}(\hat{a}_i)$

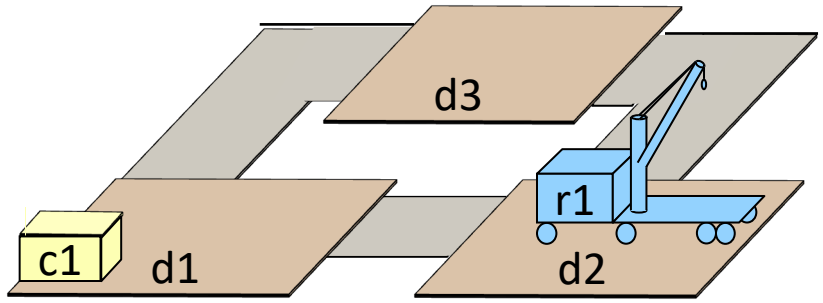
Solution extraction starting at $\hat{g}_1 = g$



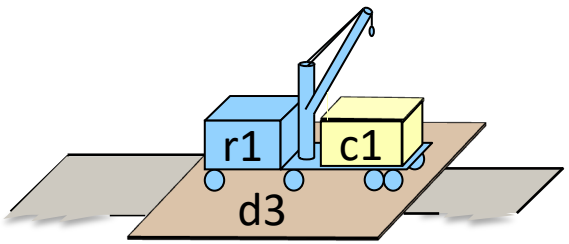
- $\hat{a}_1$ is a minimal set of actions such that $\gamma^+(\hat{s}_0,\hat{a}_1)$ r-satisfies $\hat{g}_1$
  - $\langle \hat{a}_1 \rangle$ is a minimal relaxed solution
  - two actions, each with cost 1, so $h^{FF}(s_1) = 2$

Example

- Computing $h^{FF}(s_2)$
  - 1. construct a relaxed solution
    - at each step, include all r-applicable actions



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



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

// construct a relaxed solution $\langle A_1, A_2, \ldots, A_k \rangle$:
$\hat{s}_0 \leftarrow s$
for $k = 1$ by 1 until a subset of $\hat{s}_k$ r-satisfies g
  $A_k \leftarrow$ {all actions r-applicable in $\hat{s}_{k-1}$}; $\hat{s}_k \leftarrow \gamma^+(s_{k-1}, A_k)$
  if $k > 1$ and $\hat{s}_k = \hat{s}_{k-1}$ then return ∞

RPG starting at $\hat{s}_0 = s_2$

Atoms in $\hat{s}_0$=s_2:
loc(r1) = d2
loc(c1) = d1
cargo(r1) = nil

Actions in A_1:
move(r1,d2,d3)
move(r1,d2,d1)

Atoms in $\hat{s}_1$:
loc(r1) = d3
loc(r1) = d1
loc(r1) = d2
loc(c1) = d1
cargo(r1) = nil

Actions in A_2:
move(r1,d3,d2)
move(r1,d1,d2)
move(r1,d3,d1)
move(r1,d1,d3)
move(r1,d2,d1)
move(r1,d2,d3)
load(r1,c1,d1)

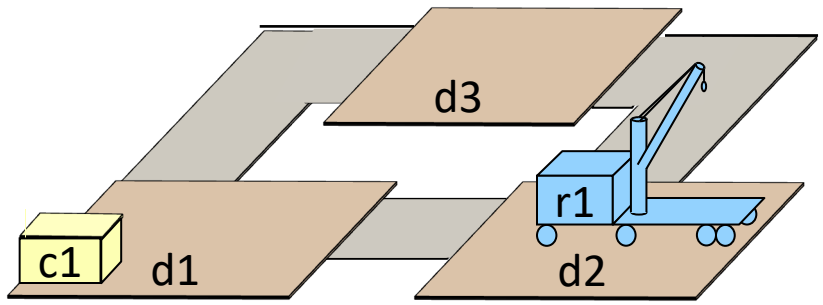
Atoms in $\hat{s}_2$:
from $\hat{s}_1$:
loc(r1) = d2
loc(c1) = d1
cargo(r1) = nil
loc(r1) = d1
loc(r1) = d3
cargo(r1) = c1
loc(c1) = r1

$\hat{s}_2$ r-satisfies g, so $\langle A_1, A_2 \rangle$ is a relaxed solution

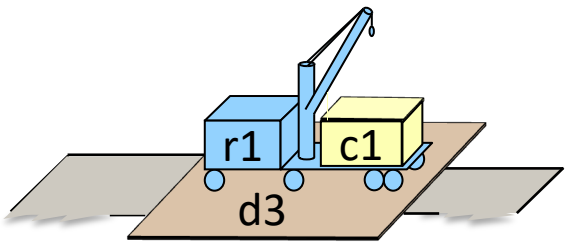
Example

- Computing $h^{FF}(s_1)$
  2. extract a *minimal* relaxed solution
     - if you remove any actions from it, it's no longer a relaxed solution

// extract minimal relaxed solution $\langle \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_k \rangle$:
$\hat{g}_k \leftarrow g$
for $i = k, k-1, \ldots, 1$:
  $\hat{a}_i \leftarrow$ any minimal subset of A_i such that $\gamma^+(\hat{s}_{i-1}, \hat{a}_i)$ r-satisfies $\hat{g}_i$
  $\hat{g}_{i-1} \leftarrow (\hat{g}_i \setminus \text{eff}(\hat{a}_i)) \cup \text{pre}(\hat{a}_i)$

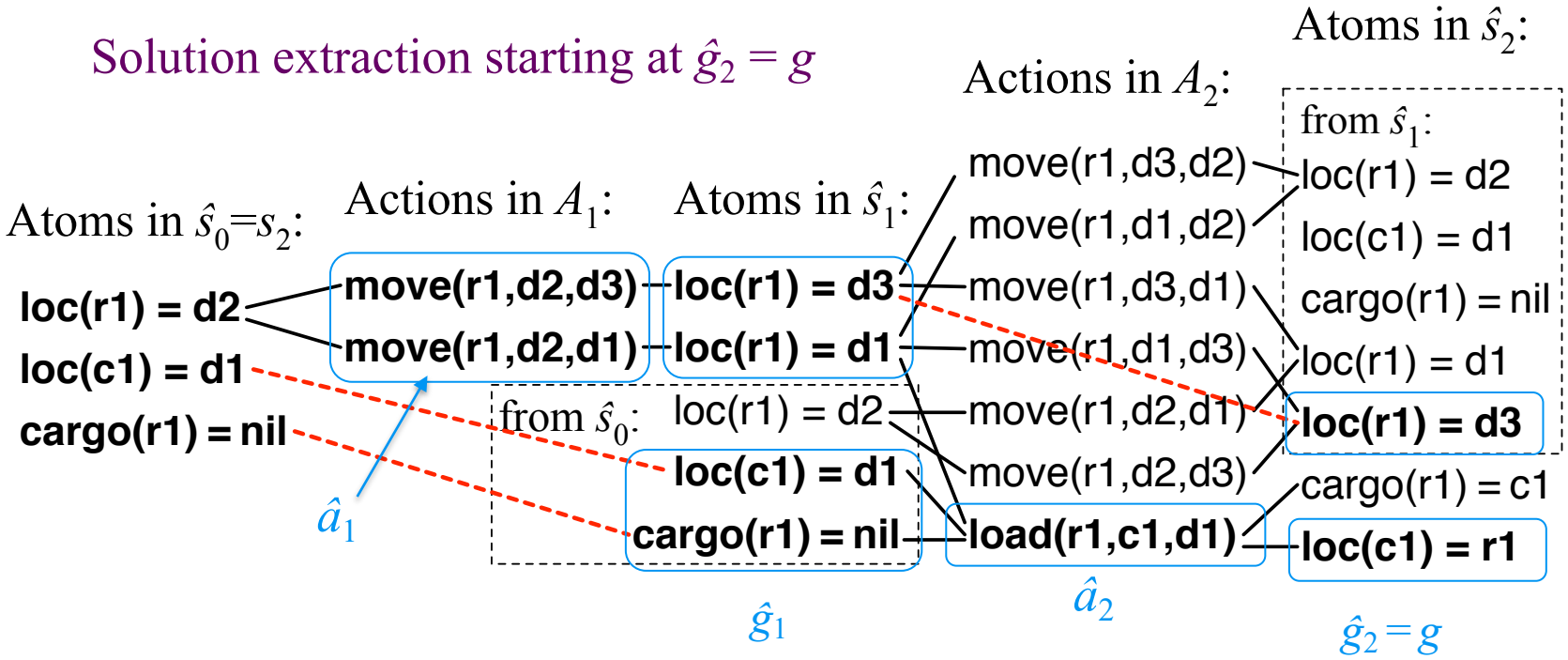


s_2 = {loc(r1)=d2, cargo(r1)=nil, loc(c1)=d2}



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

Solution extraction starting at $\hat{g}_2 = g$



- $\langle \hat{a}_1, \hat{a}_2 \rangle$ is a minimal relaxed solution
- each action's cost is 1, so $h^{FF}(s_2) = 3$

Properties

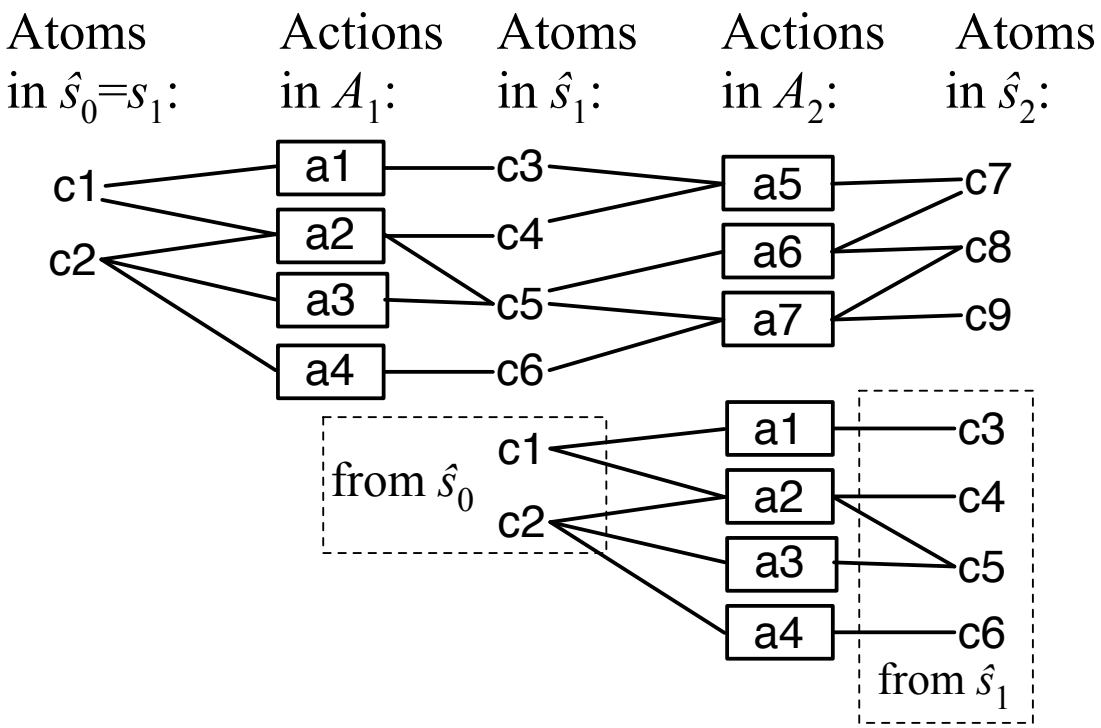
- Running time is polynomial in $|A| + \sum_{x \in X} |\text{Range}(x)|$
- $h^{\text{FF}}(s)$ = value returned by HFF(Σ,s,g)

  $= \sum_i \text{cost}(\hat{a}_i)$

  $= \sum_i \sum \{\text{cost}(a) \mid a \in \hat{a}_i\}$

  - each $\hat{a}_i$ is a minimal set of actions such that $\gamma^+(\hat{s}_{i-1},\hat{a}_i)$ r-satisfies $\hat{g}_i$
    - *minimal* doesn't mean *smallest*
- $h^{\text{FF}}(s)$ is ambiguous
  - depends on *which* minimal sets we choose
- h^{FF} not admissible
- $h^{\text{FF}}(s) \geq h^+(s)$ = *smallest* cost of any relaxed plan from s to goal

Example

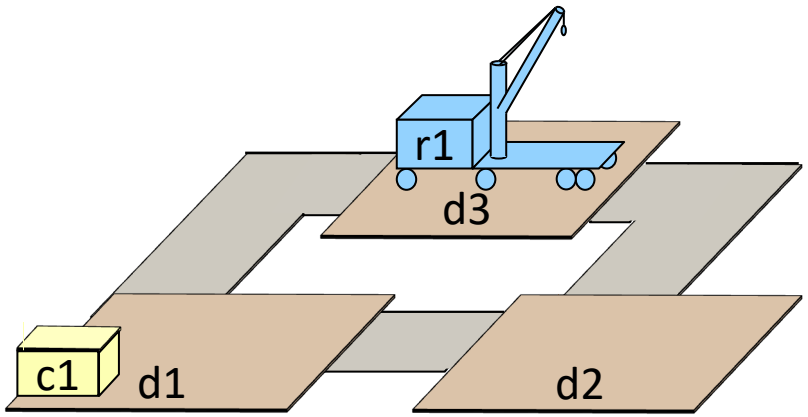


Poll. Suppose the goal atoms are c7, c8, c9. How many minimal relaxed solutions are there?

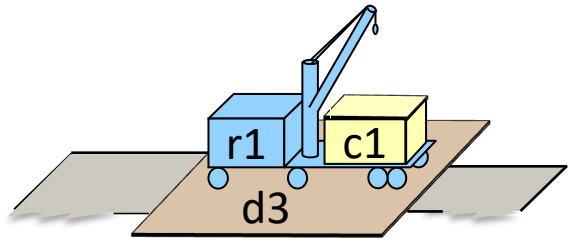
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. ≥ 8

2.3.3 Landmark Heuristics

- $P = (\Sigma, s_0, g)$ be a planning problem
- Let $\varphi = \varphi_1 \vee \ldots \vee \varphi_m$ be a disjunction of ground atoms
- φ is a *disjunctive landmark* for P if φ is true at some point in every solution for P



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



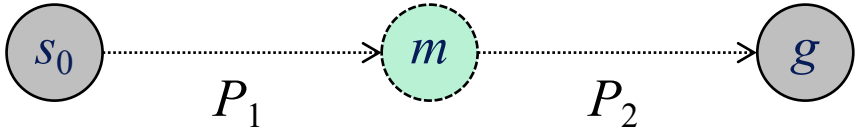
g = {loc(r1)=d3, loc(c1)=r1}

- Example disjunctive landmarks
  - loc(r1)=d1
  - loc(r1)=d3
  - loc(r1)=d3 ∨ loc(r1)=d2

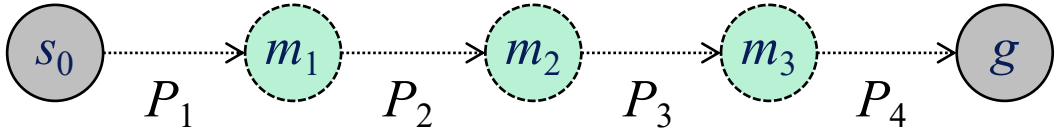
From now on, I'll abbreviate "disjunctive landmark" as "landmark"

Why are Landmarks Useful?

- Can break a problem down into smaller subproblems



- Suppose m is a landmark
  - Every solution to P must achieve m
- **Possible strategy:**
  - find a plan to go from s_0 to any state s_1 that satisfies m
  - find a plan to go from s_1 to any state s_2 that satisfies g



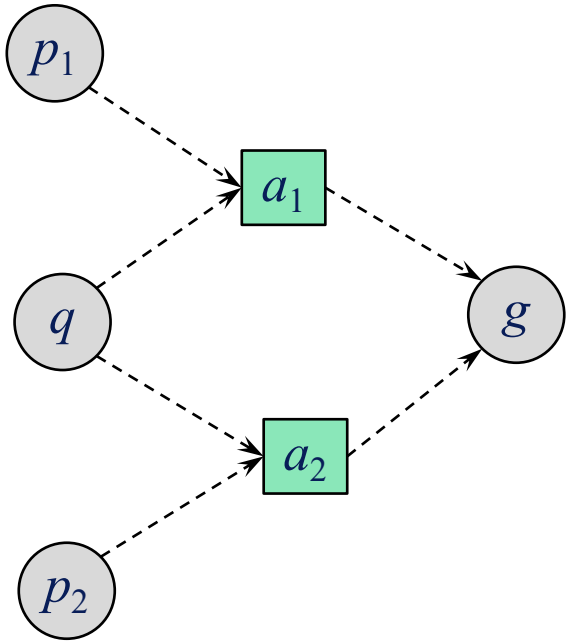
- Suppose m_1, m_2, m_3 are landmarks
  - Every solution to P must achieve m_1, then m_2, then m_3
- **Possible strategy:**
  - find a plan to go from s_0 to any state s_1 that satisfies m_1
  - find a plan to go from s_1 to any state s_2 that satisfies m_2
  - …

Computing Landmarks

- Given a formula φ
  - PSPACE-hard (worst case) to decide whether φ is a landmark
  - As hard as solving the planning problem itself
- We can't easily find *all* possible landmarks
- But there are often useful landmarks that can be found more easily
  - polynomial time
  - Going to see one such procedure based on *Relaxed Planning Graphs*
- Why Relaxed Planning Graphs?
  - Easier to solve relaxed planning problems
  - Easier to find landmarks for them
  - A landmark for a relaxed planning problem is also landmark for the original planning problem

RPG-based Landmark Computation

- **Main idea:**
  - if φ is a landmark, get new landmarks from the preconditions of the actions that achieve φ
- **Example:**
  - goal g
  - $\{a_1, a_2\}$ = all actions that achieve g
  - pre(a_1) = $\{p_1, q\}$
  - pre(a_2) = $\{p_2, q\}$
  - To achieve g, must achieve $(p_1 \wedge q) \vee (p_2 \wedge q)$
    - same as $q \wedge (p_1 \vee p_2)$
  - Landmarks:
    - q
    - $p_1 \vee p_2$



RPG-based Landmark Computation

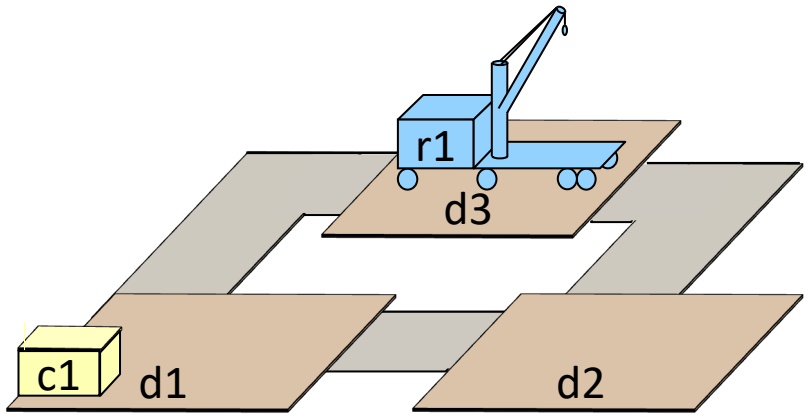
- Suppose goal is $g = \{g_1, g_2, \ldots, g_k\}$
  - Trivially, every g_i is a landmark
- Suppose g_1 = loc(r1)=d1
  - Two actions can achieve g_1: move(r1,d3,d1) and move(r1,d2,d1)
- Preconditions loc(r1)=d3 and loc(r1)=d2
- New landmark:

  φ' = loc(r1)=d3 ∨ loc(r1)=d2

move(r, d, e)
  pre: loc(r)=d
  eff: loc(r) ← e

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

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



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

RPG-based Landmark Computation

RPG-Landmarks(s_0, $g = \{g_1, g_2, \ldots, g_k\}$)

$queue \leftarrow \{g_i \in g \mid s_0$ doesn't satisfy $g_i\}$; $Landmarks \leftarrow \emptyset$

while $queue \neq \emptyset$

remove a g_i from $queue$; add it to $Landmarks$

$R \leftarrow$ {actions whose effects include g_i}

if s_0 satisfies pre(a) for some $a \in R$ then return $Landmarks$

generate RPG from s_0 using $A \setminus R$, stopping when $\hat{s}_k = \hat{s}_{k-1}$

$N \leftarrow$ {all actions in R that are r-applicable in $\hat{s}_k$}

if $N = \emptyset$ then return failure

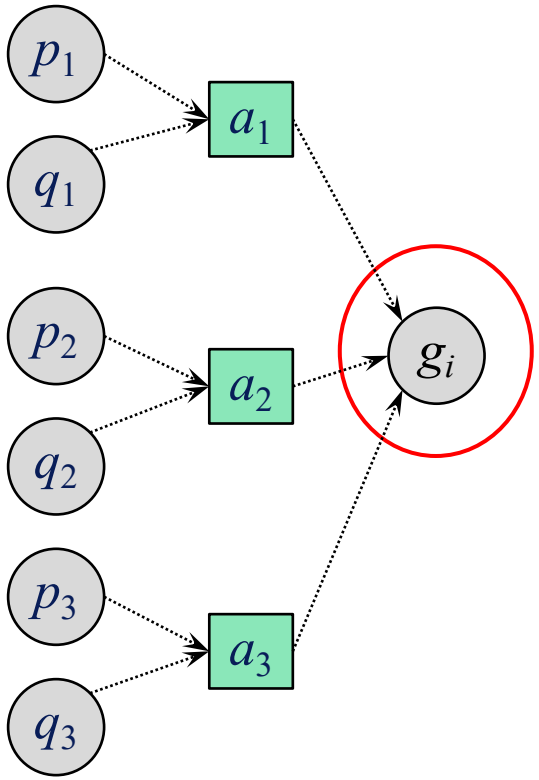
$Preconds \leftarrow \bigcup\{\text{pre}(a) \mid a \in N\} \setminus s_0$

$\Phi \leftarrow \{p_1 \vee p_2 \vee \ldots \vee p_m \mid m \leq 4$, every action in N has at least one p_i as a precondition, and every $p_i \in Preconds\}$

for each $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

add φ to $queue$

return $Landmarks$



RPG-based Landmark Computation

RPG-Landmarks(s_0, $g = \{g_1, g_2, \ldots, g_k\}$)

$queue \leftarrow \{g_i \in g \mid s_0$ doesn't satisfy $g_i\}$; $Landmarks \leftarrow \emptyset$

while $queue \neq \emptyset$

remove a g_i from $queue$; add it to $Landmarks$

$R \leftarrow$ {actions whose effects include g_i}

if s_0 satisfies pre(a) for some $a \in R$ then return $Landmarks$

generate RPG from s_0 using $A \setminus R$, stopping when $\hat{s}_k = \hat{s}_{k-1}$

$N \leftarrow$ {all actions in R that are r-applicable in $\hat{s}_k$}

if $N = \emptyset$ then return failure

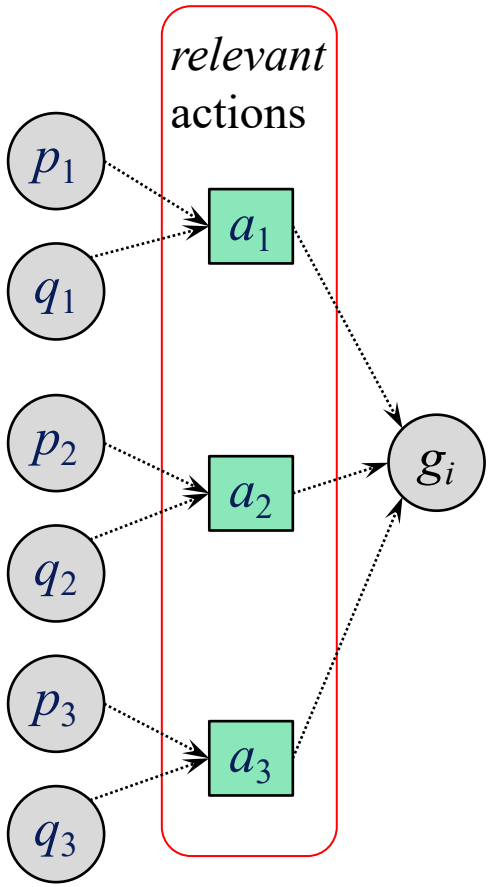
$Preconds \leftarrow \bigcup\{\text{pre}(a) \mid a \in N\} \setminus s_0$

$\Phi \leftarrow \{p_1 \vee p_2 \vee \ldots \vee p_m \mid m \leq 4$, every action in N has at least one p_i as a precondition, and every $p_i \in Preconds\}$

for each $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

add φ to $queue$

return $Landmarks$



RPG-based Landmark Computation

RPG-Landmarks(s_0, $g = \{g_1, g_2, \ldots, g_k\}$)

queue ← $\{g_i \in g \mid s_0$ doesn't satisfy $g_i\}$; *Landmarks* ← ∅

while *queue* ≠ ∅

remove a g_i from *queue*; add it to *Landmarks*

R ← {actions whose effects include g_i}

if s_0 satisfies pre(a) for some $a \in R$ then return *Landmarks*

generate RPG from s_0 using $A \setminus R$, stopping when $\hat{s}_k = \hat{s}_{k-1}$

N ← {all actions in R that are r-applicable in $\hat{s}_k$}

if $N = \emptyset$ then return failure

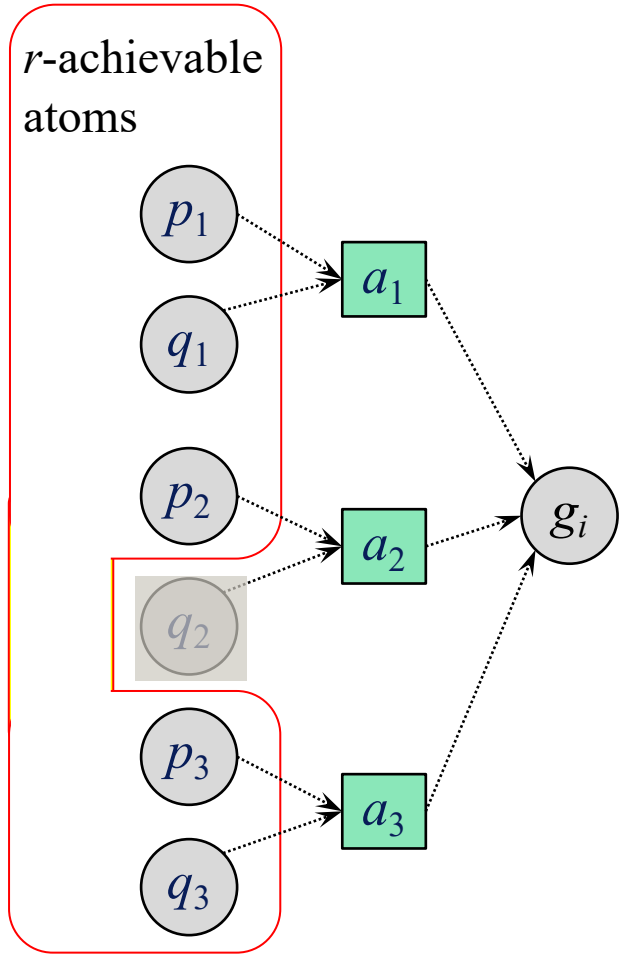
Preconds ← $\bigcup\{\text{pre}(a) \mid a \in N\} \setminus s_0$

$\Phi \leftarrow \{p_1 \vee p_2 \vee \ldots \vee p_m \mid m \leq 4$, every action in N has at least one p_i as a precondition, and every $p_i \in$ *Preconds*$\}$

for each $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

add φ to *queue*

return *Landmarks*



RPG-based Landmark Computation

RPG-Landmarks(s_0, $g = \{g_1, g_2, \ldots, g_k\}$)

  $queue \leftarrow \{g_i \in g \mid s_0$ doesn't satisfy $g_i\}$; $Landmarks \leftarrow \emptyset$

  while $queue \neq \emptyset$

    remove a g_i from $queue$; add it to $Landmarks$

    $R \leftarrow$ {actions whose effects include g_i}

    if s_0 satisfies pre(a) for some $a \in R$ then return $Landmarks$

    generate RPG from s_0 using $A \setminus R$, stopping when $\hat{s}_k = \hat{s}_{k-1}$

    $N \leftarrow$ {all actions in R that are r-applicable in $\hat{s}_k$}

    if $N = \emptyset$ then return failure

    $Preconds \leftarrow \bigcup\{\text{pre}(a) \mid a \in N\} \setminus s_0$

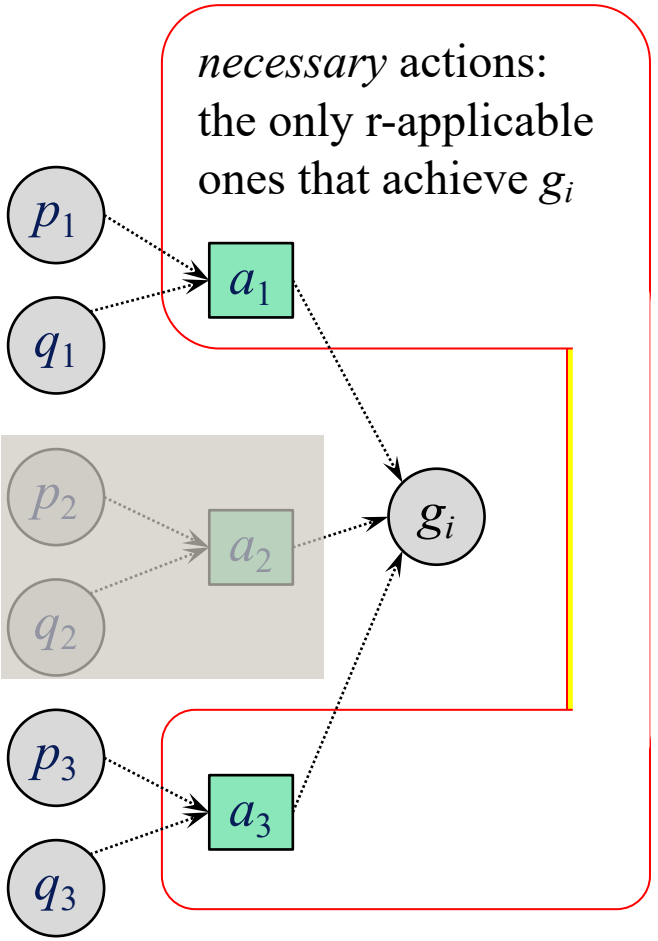
    $\Phi \leftarrow \{p_1 \vee p_2 \vee \ldots \vee p_m \mid m \leq 4$, every action in N has at least one p_i as a precondition, and every $p_i \in Preconds\}$

    for each $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

      add φ to $queue$

  return $Landmarks$

$Preconds = \{p_1, q_1, p_3, q_3\}$



RPG-based Landmark Computation

RPG-Landmarks(s_0, $g = \{g_1, g_2, \ldots, g_k\}$)

queue ← $\{g_i \in g \mid s_0$ doesn't satisfy $g_i\}$; *Landmarks* ← ∅

while *queue* ≠ ∅

remove a g_i from *queue*; add it to *Landmarks*

R ← {actions whose effects include g_i}

if s_0 satisfies pre(a) for some $a \in R$ then return *Landmarks*

generate RPG from s_0 using $A \setminus R$, stopping when $\hat{s}_k = \hat{s}_{k-1}$

N ← {all actions in R that are r-applicable in $\hat{s}_k$}

if $N = \emptyset$ then return failure

Preconds ← $\bigcup\{\text{pre}(a) \mid a \in N\} \setminus s_0$

$\Phi \leftarrow \{p_1 \vee p_2 \vee \ldots \vee p_m \mid m \leq 4$, every action in N has at least one p_i as a precondition, and every $p_i \in$ *Preconds*$\}$

for each $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

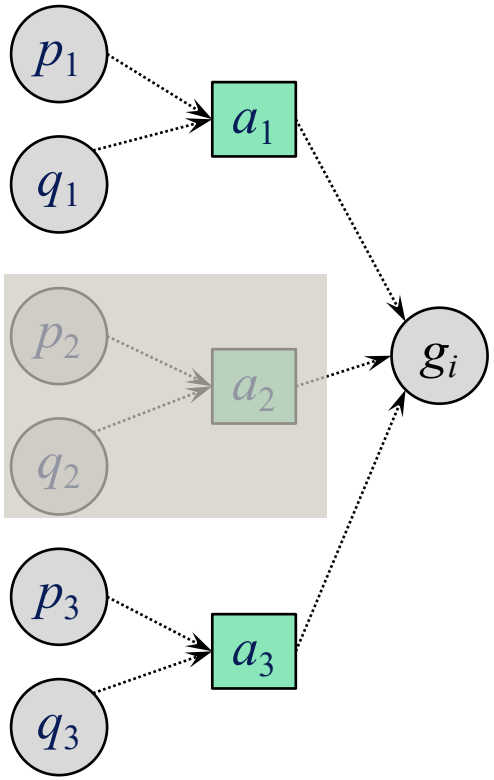
add φ to *queue*

return *Landmarks*

Not in book

$\Phi = \{p_1 \vee p_3,\ p_1 \vee q_3,\ q_1 \vee p_3,\ q_1 \vee q_3,\ p_1 \vee q_1 \vee p_3,\ p_1 \vee q_1 \vee q_3,\ p_1 \vee p_3 \vee q_3,\ q_1 \vee p_3 \vee q_3,\ p_1 \vee q_1 \vee p_3 \vee q_3\}$

queue = $\langle p_1 \vee p_3,\ p_1 \vee q_3,\ q_1 \vee p_3,\ q_1 \vee q_3 \rangle$



Example

RPG-Landmarks(s_0, $g = \{g_1, g_2, \ldots, g_k\}$)

queue ← $\{g_i \in g \mid s_0$ doesn't satisfy $g_i\}$; *Landmarks* ← ∅

while *queue* ≠ ∅

remove a g_i from *queue*; add it to *Landmarks*

R ← {actions whose effects include g_i}

if s_0 satisfies pre(a) for some $a \in R$ then return *Landmarks*

generate RPG from s_0 using $A \setminus R$, stopping when $\hat{s}_k = \hat{s}_{k-1}$

N ← {all actions in R that are r-applicable in $\hat{s}_k$}

if $N = \emptyset$ then return failure

Preconds ← $\bigcup\{\text{pre}(a) \mid a \in N\} \setminus s_0$

$\Phi \leftarrow \{p_1 \vee p_2 \vee \ldots \vee p_m \mid m \leq 4$, every action in N has at least one p_i as a precondition, and every $p_i \in$ *Preconds*$\}$

for each $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

add φ to *queue*

return *Landmarks*

queue = ⟨loc(c1)=r1⟩

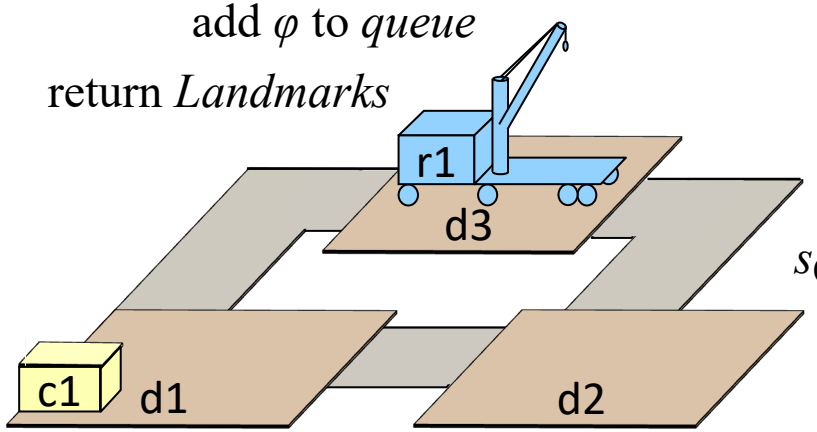
Landmarks = ∅

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

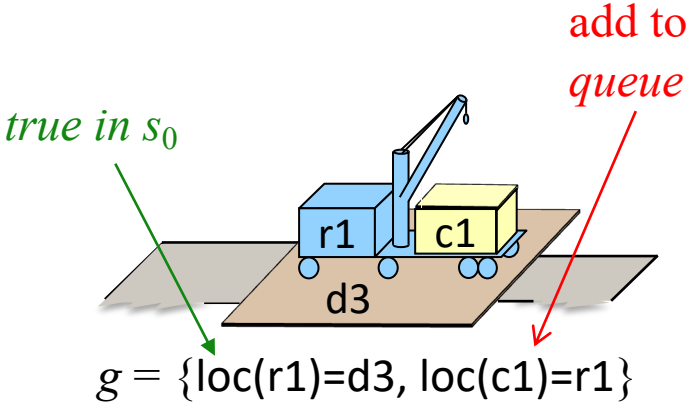
move(r, d, e)
- pre: loc(r)=d
- eff: loc(r)←e

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

$r \in$ *Robots*
$c \in$ *Containers*
$l,d,e \in$ *Locs*



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



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

Example

RPG-Landmarks(s_0, $g = \{g_1, g_2, \ldots, g_k\}$)

$queue \leftarrow \{g_i \in g \mid s_0$ doesn't satisfy $g_i\}$; $Landmarks \leftarrow \emptyset$

while $queue \neq \emptyset$

remove a g_i from $queue$; add it to $Landmarks$

$R \leftarrow$ {actions whose effects include g_i}

if s_0 satisfies pre(a) for some $a \in R$ then return $Landmarks$

generate RPG from s_0 using $A \setminus R$, stopping when $\hat{s}_k = \hat{s}_{k-1}$

$N \leftarrow$ {all actions in R that are r-applicable in $\hat{s}_k$}

if $N = \emptyset$ then return failure

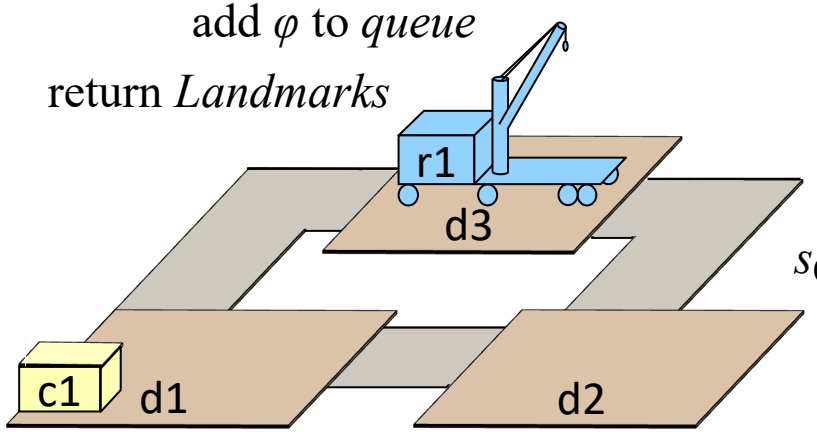
$Preconds \leftarrow \bigcup\{\text{pre}(a) \mid a \in N\} \setminus s_0$

$\Phi \leftarrow \{p_1 \vee p_2 \vee \ldots \vee p_m \mid m \leq 4$, every action in N has at least one p_i as a precondition, and every $p_i \in Preconds\}$

for each $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

add φ to $queue$

return $Landmarks$



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

queue = ⟨⟩

Landmarks = {loc(c1)=r1}

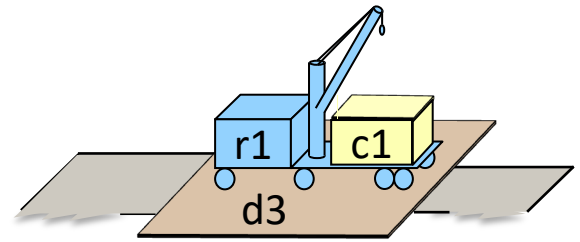
R = {load(r1,c1,d1), load(r1,c1,d2), load(r1,c1,d3)}

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

move(r, d, e)
- pre: loc(r)=d
- eff: loc(r)←e

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

$r \in Robots$
$c \in Containers$
$l,d,e \in Locs$



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

Example

RPG-Landmarks(s_0, $g = \{g_1, g_2, \ldots, g_k\}$)

queue ← $\{g_i \in g \mid s_0$ doesn't satisfy $g_i\}$; *Landmarks* ← ∅

while *queue* ≠ ∅

remove a g_i from *queue*; add it to *Landmarks*

R ← {actions whose effects include g_i}

if s_0 satisfies pre(a) for some $a \in R$ then return *Landmarks*

generate RPG from s_0 using $A \setminus R$, stopping when $\hat{s}_k = \hat{s}_{k-1}$

N ← {all actions in R that are r-applicable in $\hat{s}_k$}

if $N = \emptyset$ then return failure

Preconds ← $\bigcup\{\text{pre}(a) \mid a \in N\} \setminus s_0$

$\Phi \leftarrow \{p_1 \vee p_2 \vee \ldots \vee p_m \mid m \le 4$, every action in N has at least one p_i as a precondition, and every $p_i \in$ *Preconds*$\}$

for each $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

add φ to *queue*

return *Landmarks*

queue = ⟨⟩

Landmarks = {loc(c1)=r1}

R = {load(r1,c1,d1), load(r1,c1,d2), load(r1,c1,d3)}

N = {load(r1,c1,d1)}

load (r1,c1,d1)

pre: cargo(r1)=nil, loc(c1)=d1, loc(r1)=d1 — in $\hat{s}_2$

load(r, c, l)

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

eff: cargo(r)←c, loc(c)←r

move(r, d, e)

pre: loc(r)=d

eff: loc(r)←e

unload(r, c, l)

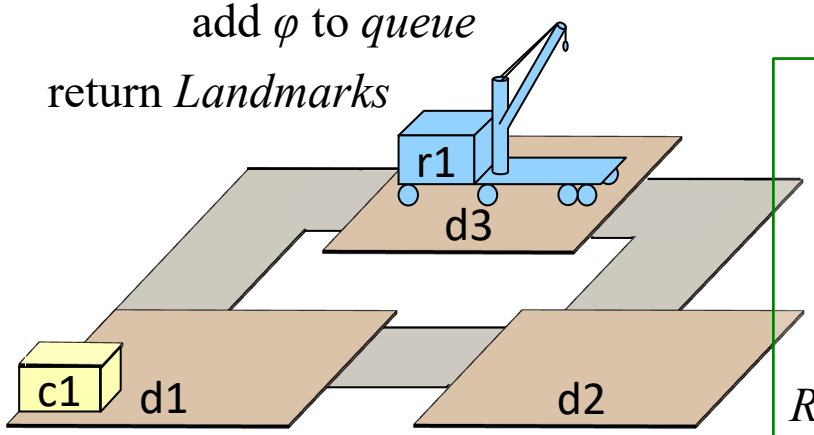
pre: loc(c)=r, loc(r)=l

eff: cargo(r)←nil, loc(c)←l

$r \in$ *Robots*

$c \in$ *Containers*

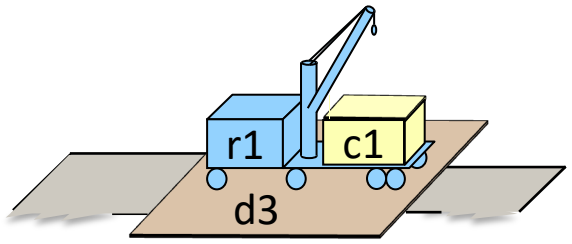
$l,d,e \in$ *Locs*



$\hat{s}_0$:	A_1:	both $\hat{s}_1$ and $\hat{s}_2$:
loc(c1)=d1	move(r1,d3,d1)	loc(r1)=d1
loc(r1)=d3	move(r1,d3,d2)	loc(r1)=d2
cargo(r1)=nil		

From $\hat{s}_0$: loc(c1)=d1, loc(r1)=d3, cargo(r1)=nil

Relaxed planning graph using $A \setminus R$



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

Example

RPG-Landmarks(s_0, $g = \{g_1, g_2, \ldots, g_k\}$)

queue ← $\{g_i \in g \mid s_0$ doesn't satisfy $g_i\}$; *Landmarks* ← ∅

while *queue* ≠ ∅

remove a g_i from *queue*; add it to *Landmarks*

$R \leftarrow$ {actions whose effects include g_i}

if s_0 satisfies pre(a) for some $a \in R$ then return *Landmarks*

generate RPG from s_0 using $A \setminus R$, stopping when $\hat{s}_k = \hat{s}_{k-1}$

$N \leftarrow$ {all actions in R that are r-applicable in $\hat{s}_k$}

if $N = \emptyset$ then return failure

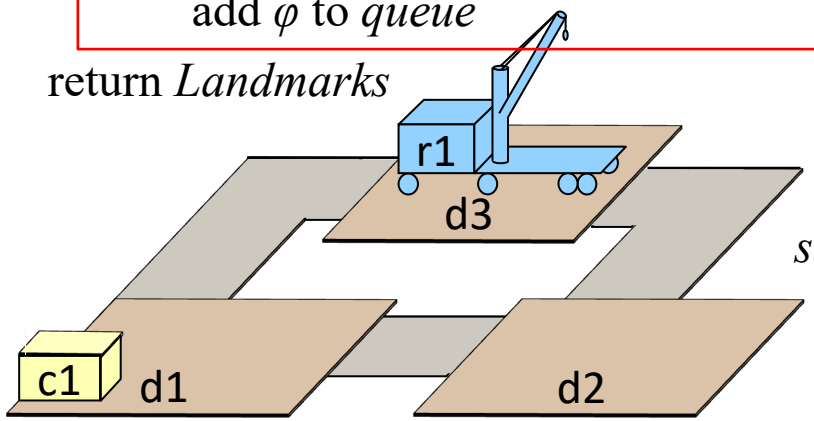
Preconds ← $\bigcup\{\text{pre}(a) \mid a \in N\} \setminus s_0$

$\Phi \leftarrow \{p_1 \vee p_2 \vee \ldots \vee p_m \mid m \le 4$, every action in N has at least one p_i as a precondition, and every $p_i \in$ *Preconds*$\}$

for each $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

add φ to *queue*

return *Landmarks*



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

~~*queue* = ⟨⟩~~

Landmarks = {loc(c1)=r1}

R = {load(r1,c1,d1),
load(r1,c1,d2),
load(r1,c1,d3)}

N = {load(r1,c1,d1)}

load (r1,c1,d1)
pre: cargo(r1)=nil, loc(c1)=d1, loc(r1)=d1 — in s_0

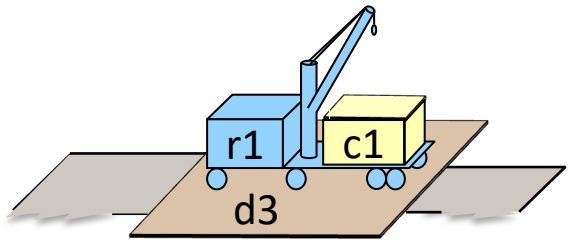
Preconds = {loc(r1)=d1}
Φ = {loc(r1)=d1}
queue = ⟨loc(r1)=d1⟩

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

move(r, d, e)
pre: loc(r)=d
eff: loc(r)←e

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

$r \in$ *Robots*
$c \in$ *Containers*
$l,d,e \in$ *Locs*



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

Example

RPG-Landmarks(s_0, $g = \{g_1, g_2, \ldots, g_k\}$)

queue ← $\{g_i \in g \mid s_0$ doesn't satisfy $g_i\}$; *Landmarks* ← ∅

while *queue* ≠ ∅

remove a g_i from *queue*; add it to *Landmarks*

R ← {actions whose effects include g_i}

if s_0 satisfies pre(a) for some $a \in R$ then return *Landmarks*

generate RPG from s_0 using $A \setminus R$, stopping when $\hat{s}_k = \hat{s}_{k-1}$

N ← {all actions in R that are r-applicable in $\hat{s}_k$}

if $N = \emptyset$ then return failure

Preconds ← $\bigcup\{\text{pre}(a) \mid a \in N\} \setminus s_0$

$\Phi \leftarrow \{p_1 \vee p_2 \vee \ldots \vee p_m \mid m \leq 4$, every action in N has at least one p_i as a precondition, and every $p_i \in$ *Preconds*$\}$

for each $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

add φ to *queue*

return *Landmarks*

queue = ⟨loc(r1)=d1⟩

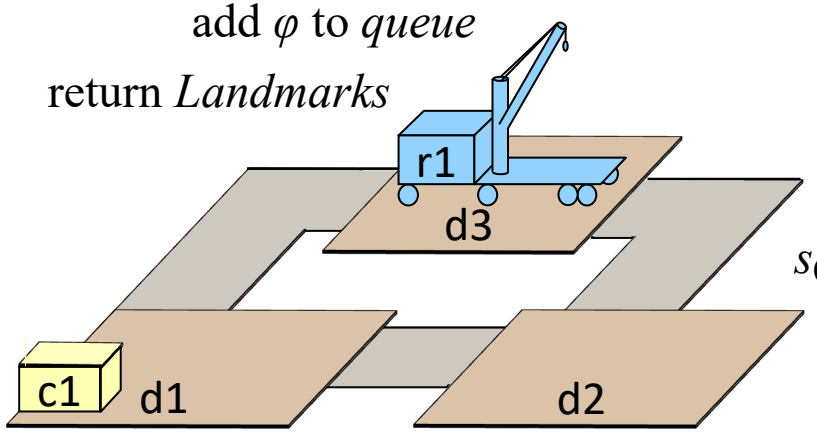
Landmarks = {loc(c1)=r1}

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

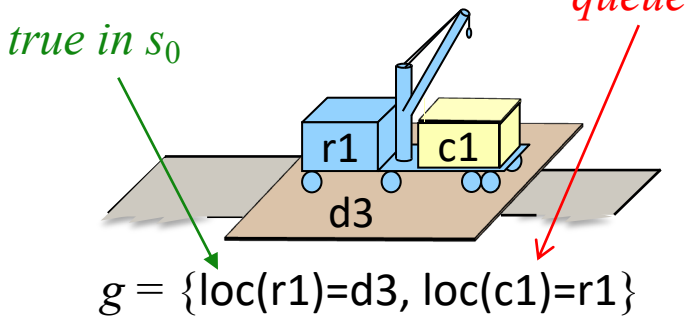
move(r, d, e)
- pre: loc(r)=d
- eff: loc(r)←e

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

$r \in$ *Robots*
$c \in$ *Containers*
$l,d,e \in$ *Locs*



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



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

Example

RPG-Landmarks(s_0, $g = \{g_1, g_2, \ldots, g_k\}$)

queue ← $\{g_i \in g \mid s_0$ doesn't satisfy $g_i\}$; *Landmarks* ← ∅

while *queue* ≠ ∅

remove a g_i from *queue*; add it to *Landmarks*

R ← {actions whose effects include g_i}

if s_0 satisfies pre(a) for some $a \in R$ then return *Landmarks*

generate RPG from s_0 using $A \setminus R$, stopping when $\hat{s}_k = \hat{s}_{k-1}$

N ← {all actions in R that are r-applicable in $\hat{s}_k$}

if $N = \emptyset$ then return failure

Preconds ← $\bigcup\{\text{pre}(a) \mid a \in N\} \setminus s_0$

$\Phi \leftarrow \{p_1 \vee p_2 \vee \ldots \vee p_m \mid m \leq 4$, every action in N has at least one p_i as a precondition, and every $p_i \in$ *Preconds*$\}$

for each $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

add φ to *queue*

return *Landmarks*

queue = ⟨⟩

Landmarks = {loc(c1)=r1, loc(r1)=d1}

R = {move(r1,d2,d1), move(r1,d3,d1)}

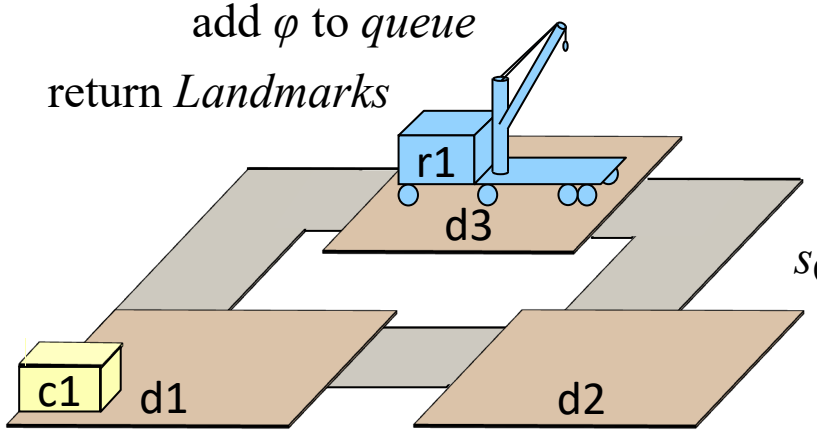
s_0 satisfies pre(move(r1,d3,d1))

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

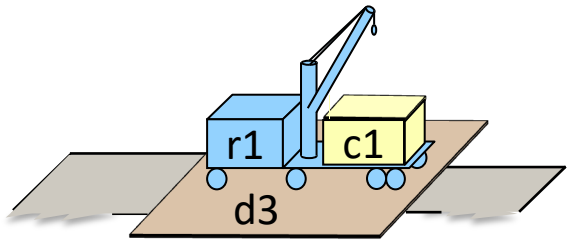
move(r, d, e)
- pre: loc(r)=d
- eff: loc(r)←e

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

$r \in$ *Robots*
$c \in$ *Containers*
$l,d,e \in$ *Locs*



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



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

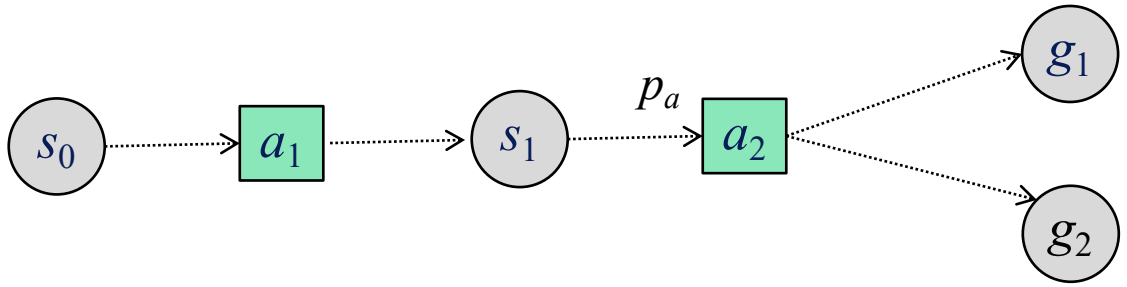
Landmark Heuristic

- Every solution to the problem needs to achieve all the computed landmarks
- One possible heuristic:
  - $h^{sl}(s)$ = number of landmarks returned by RPG-Landmarks
- Is this heuristic admissible?

Landmark Heuristic

- Every solution to the problem needs to achieve all the computed landmarks
- One possible heuristic:
  - $h^{sl}(s)$ = number of landmarks returned by RPG-Landmarks

- Not admissible



$g = \{g_1, g_2\}$
Three landmarks: g_1 , g_2 , p_a
Optimal plan: $\langle a_1, a_2 \rangle$, length = 2

- There are other more-advanced landmark heuristics
  - Some of them are admissible
  - Check textbook for references

Summary

- 2.3 Heuristic Functions
  - Straight-line distance example
  - Delete-relaxation heuristics
    - relaxed states, γ^+, h^+, HFF, h^{FF}
  - Disjunctive landmarks, RPG-Landmark, h^{sl}
    - Get necessary actions by making RPG for all non-relevant actions

Outline

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

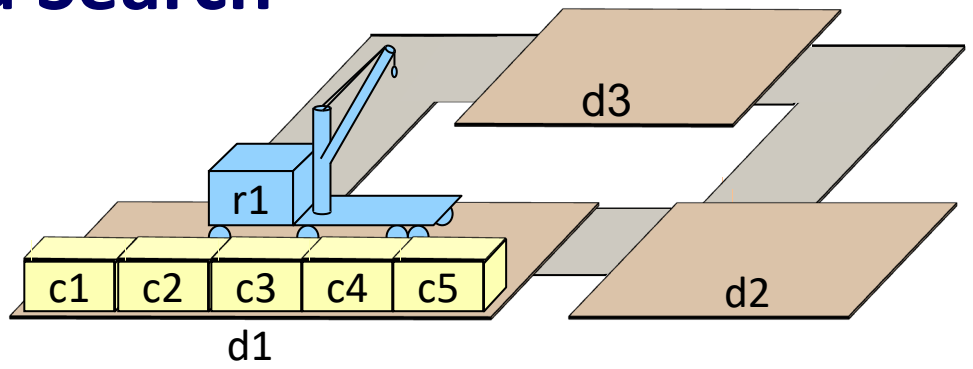
- 2.1 State-variable representation
- — Comparison with PDDL
- 2.2 Forward state-space search
- 2.6 Incorporating planning into an actor

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

- 2.3 Heuristic functions
- *Next* → 2.4 Backward search
- 2.5 Plan-space search

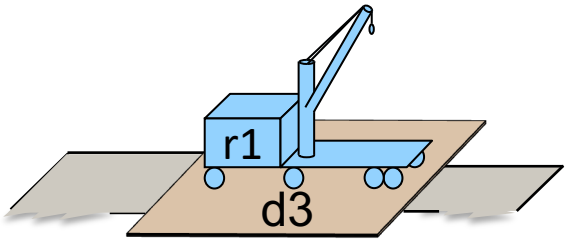
2.4 Backward Search

- 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}

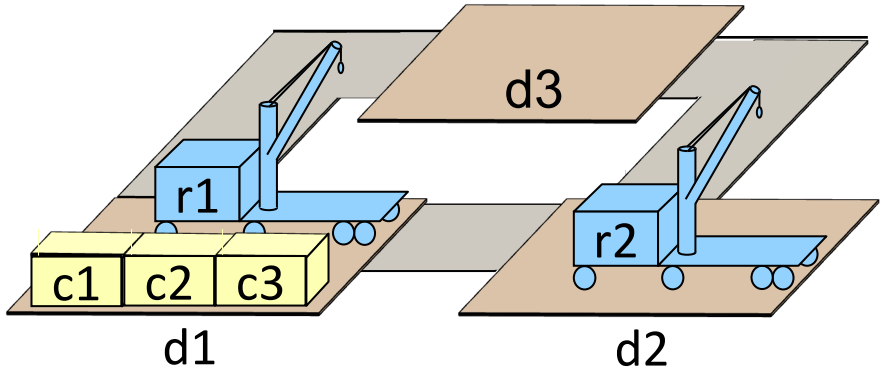


- Forward: 7 applicable actions
  - five load actions, two move actions
- Backward: g = {loc(r1)=d3}
  - two relevant actions: move(r1,d1,d3), move(r1,d2,d3)

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

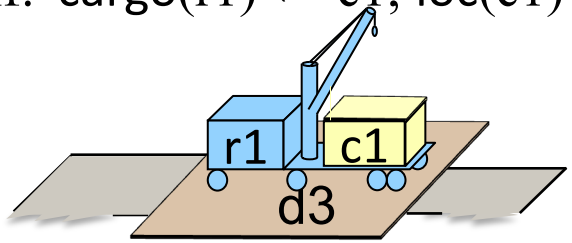
- a is *relevant* for $g = \{x_1=c_1, x_2=c_2, \ldots, x_k=c_k\}$ if
  - at least one atom in g is also in eff(a)
    - e.g., eff(a) contains $x_1 \leftarrow c_1$
  - eff(a) doesn't make any atom in g false
    - e.g., eff(a) doesn't contain $x_2 \leftarrow c'$ (where $c' \neq c_2$)
  - whenever pre(a)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



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

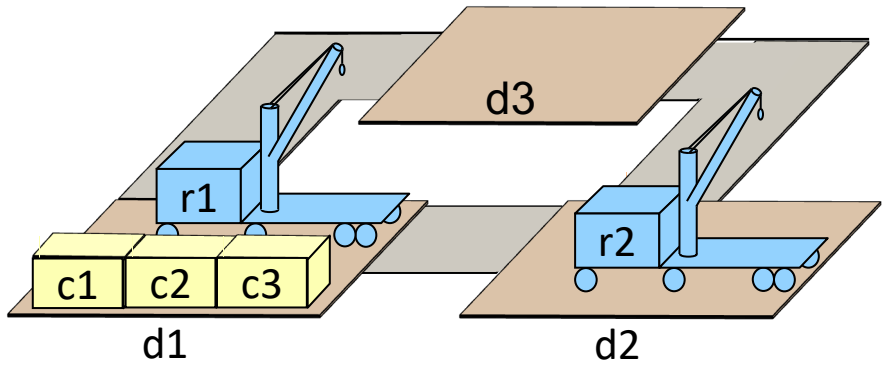
load(r,c,l)
pre: cargo(r)=nil, loc(r)=l, loc(c)=l
eff: cargo(r) ← c, loc(c) ← r

load(r1,c1,d3)
pre: cargo(r1)=nil, loc(r1)=d3, loc(c1)=d3
eff: cargo(r1) ← c1, loc(c1) ← r1



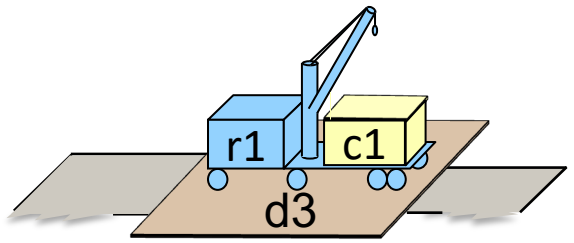
g = {cargo(r1)=c1, loc(r1)=d3}

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) ← m

load(r,c,l)
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

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

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

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

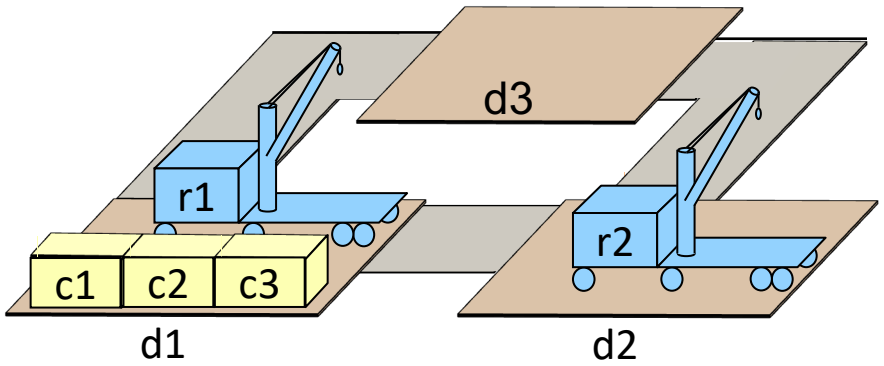
Poll: for each action below, is it relevant for g?

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



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

load(r,c,l)
  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

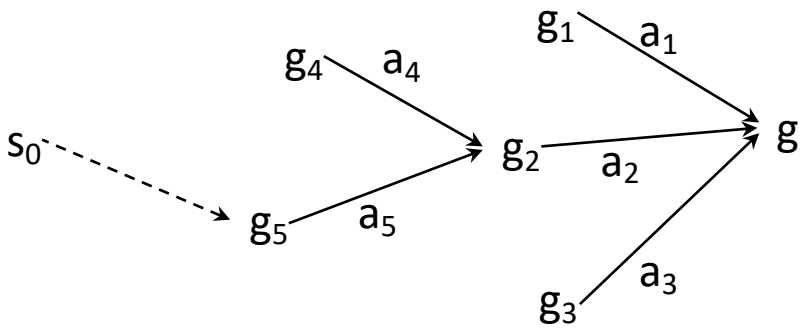
Range(r) = *Robots*

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

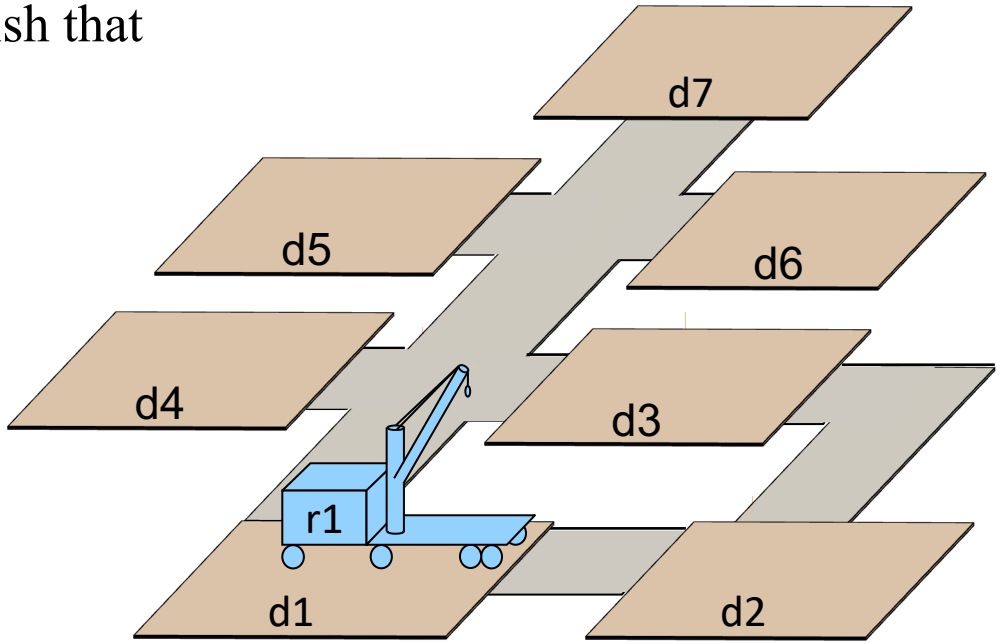
    return failure

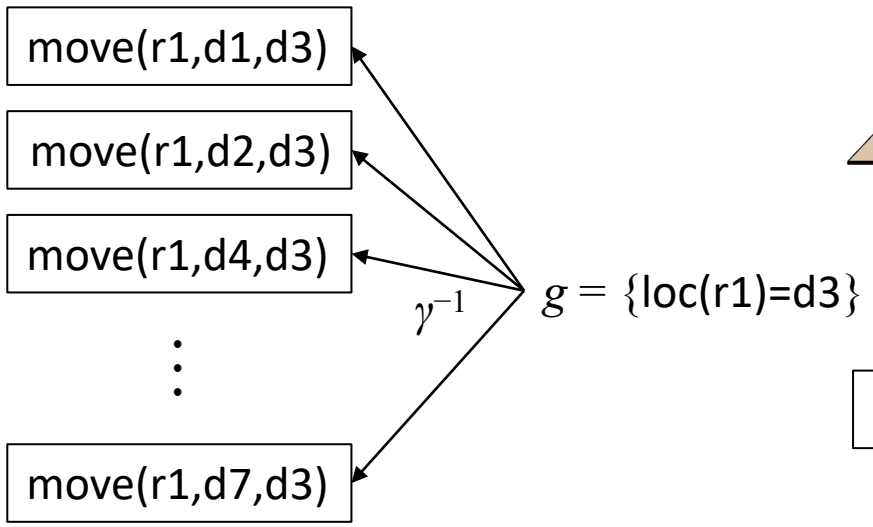
  $Visited \leftarrow Visited \cup \{g\}$

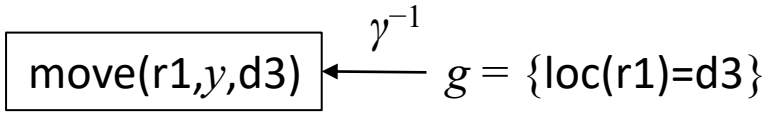
- With cycle checking, sound and complete
  - If (Σ, s_0, g_0) is solvable, then at least one execution trace will find a solution

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







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)
  $\pi \leftarrow \langle\rangle$; $g \leftarrow g_0$
  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 $a \in A'$
    $g \leftarrow \gamma^{-1}(g, a)$
    $\pi \leftarrow a.\pi$

For classical planning, this can be simplified

Lifted-backward-search$(\mathcal{A}, s_0, g)$
  $\pi \leftarrow$ the empty plan
  loop
    if s_0 satisfies g then return π
    $A \leftarrow \{(o, \theta) | o$ is a standardization of an action template in $\mathcal{A}$,
      θ is an mgu for an atom of g and an atom of $\text{effects}^+(o)$,
      and $\gamma^{-1}(\theta(g), \theta(o))$ 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, 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

2.4 Backward search

Next ⟶ 2.5 Plan-space search

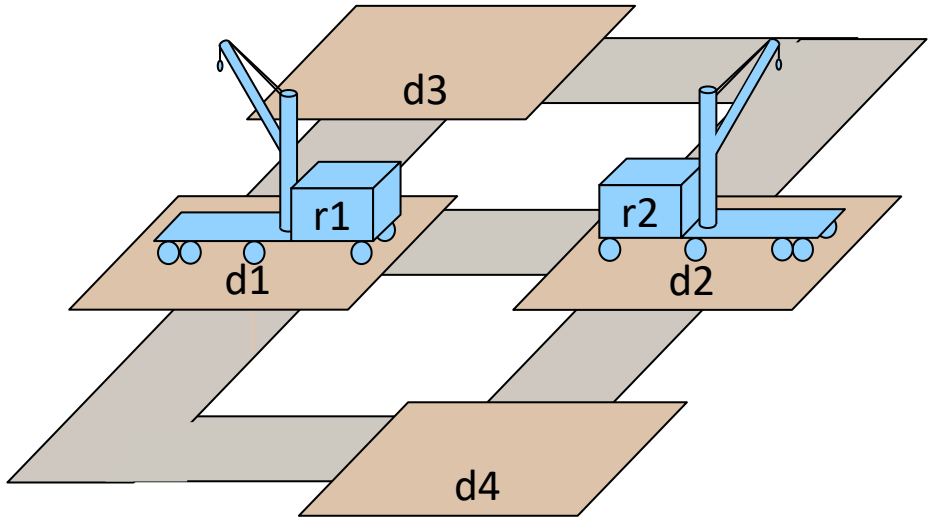
2.5 Plan-Space Search

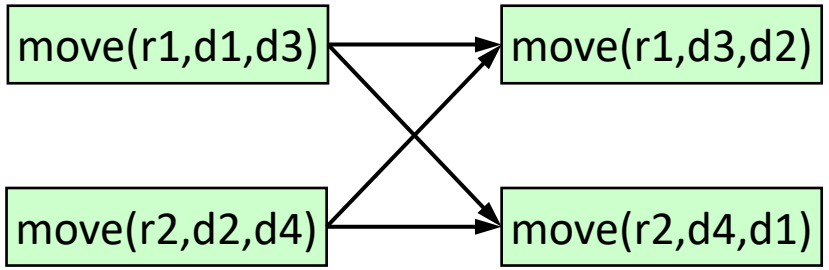
- Formulate planning as a constraint satisfaction problem
  - Use constraint-satisfaction techniques to get solutions that are more flexible than ordinary plans
    - E.g., plans in which the actions are partially ordered
    - Postpone ordering decisions until the plan is being executed
      - the 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*
  - partially ordered plan π such that every total ordering of π is a solution





arrows represent precedence

move(r, d, d')

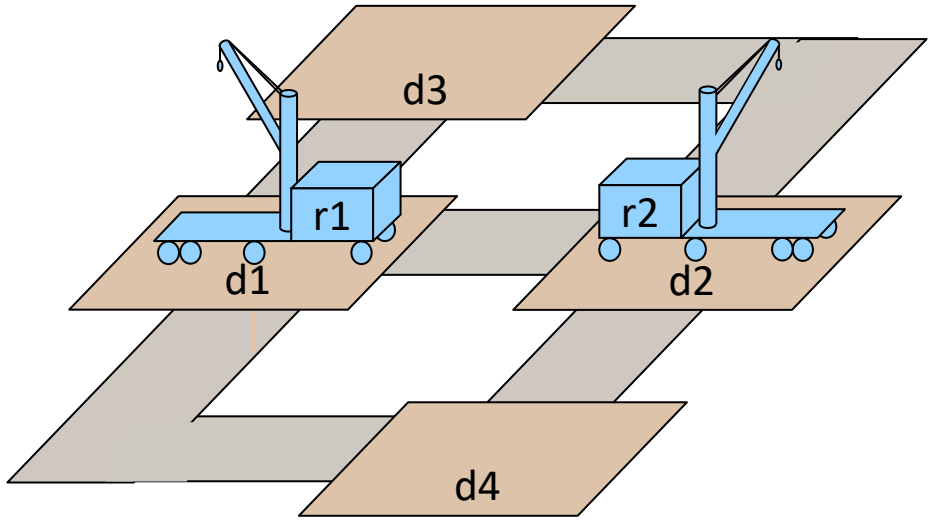
pre: loc(r) = d, occupied(d') = nil

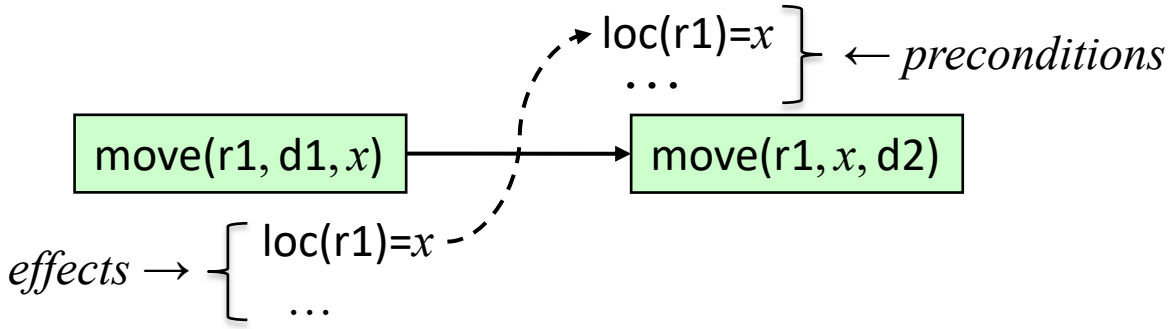
eff: loc(r) ← d', occupied(d') = r, occupied(d) = nil

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

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*





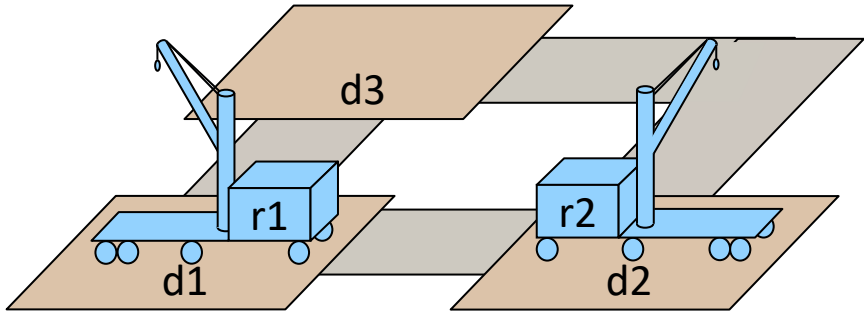
move(r, d, d')

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

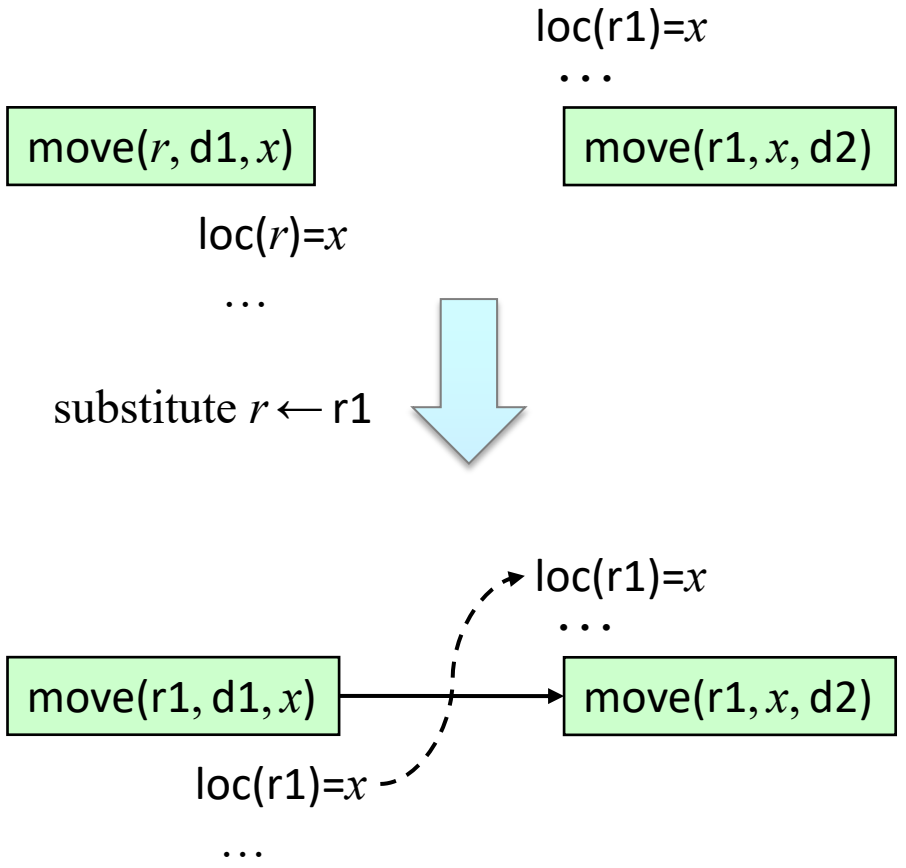
eff: loc(r) ← d', occupied(d') = r, occupied(d) = nil

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

Flaws: 1. Open Goals



- 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



move(r, d, d')

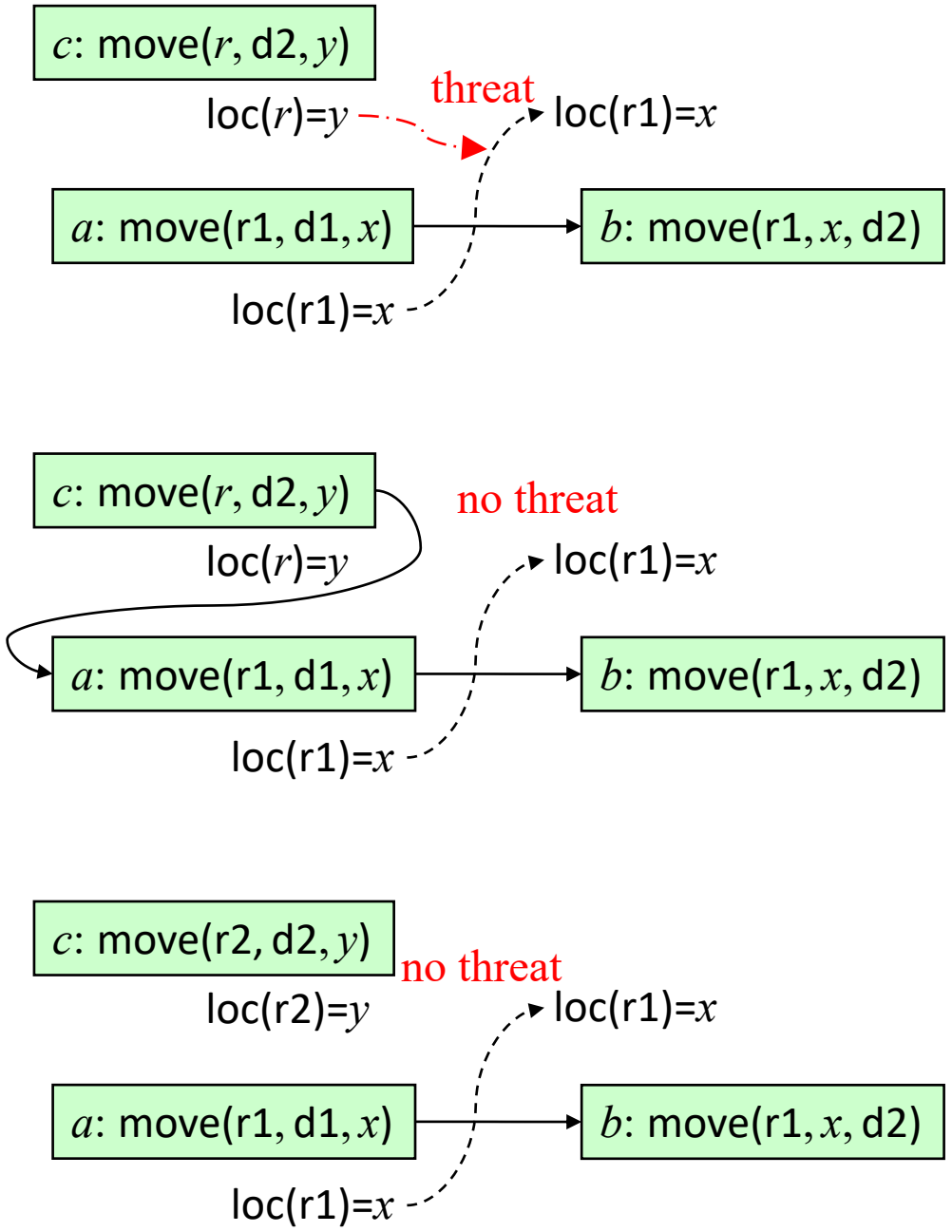
pre: loc(r) = d, occupied(d') = nil

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

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

Flaws: 2. Threats

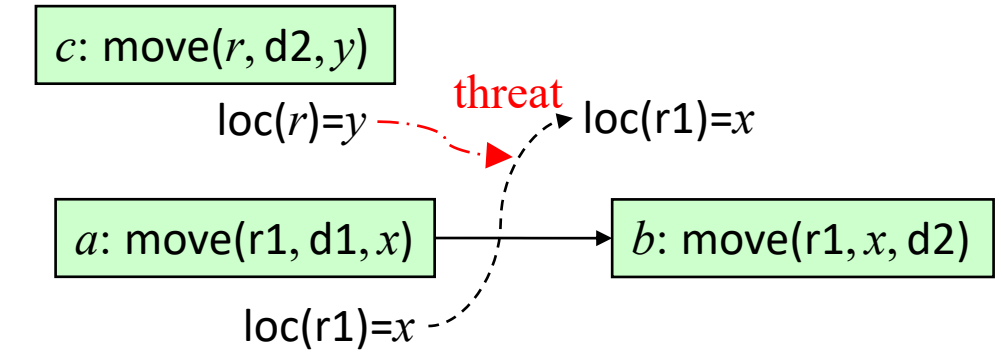
- Let l be a causal link from an effect of action a to a precondition p of action b
- Action c *threatens* l if c may come between a and b and c may affect p
  - "c may come between a and b" means the plan's current ordering constraints don't prevent it
    - plan doesn't already have $c \prec a$ or $c \prec a$
  - "c may affect p" means
    - can substitute values for variables such that c's effects either make p true or make p false
- Note: c is a threat even if it makes p true
  - l says a must be the action that establishes p for b
    - a doesn't do that if $a \prec c \prec b$ and $p \in \text{eff}(c)$
  - Plans in which c establishes p will be explored elsewhere in the search space

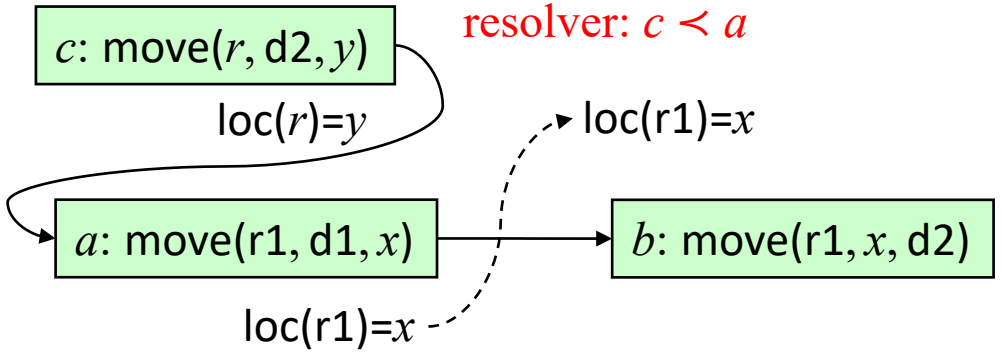


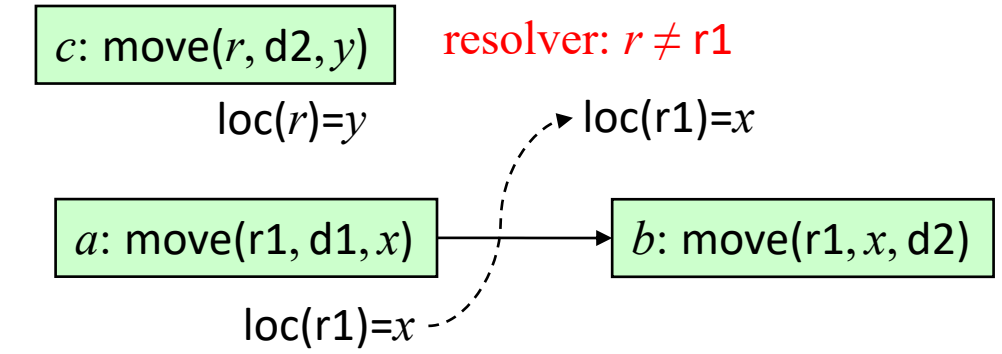
Resolving Threats

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

- Three possible resolvers:
  - Make $c \prec a$
    - applicable if the plan doesn't make $a \prec c$
  - Make $b \prec c$
    - applicable if the plan doesn't make $c \prec b$
  - Add inequality constraints that prevent c from affecting p
    - applicable if such constraints exist

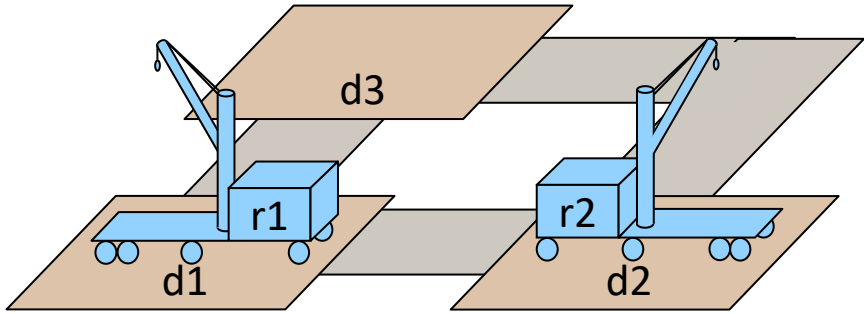






PSP Algorithm

```
PSP(Σ, π)
  loop
    if Flaws(π) = ∅ then return π
    arbitrarily select f ∈ Flaws(π)
    R ← {all feasible resolvers for f}
    if R = ∅ then return failure
    nondeterministically choose ρ ∈ R
    π ← ρ(π)
  return π
```



- 2 open goals
- no threats

select → loc(r1) = d2
loc(r2) = d1

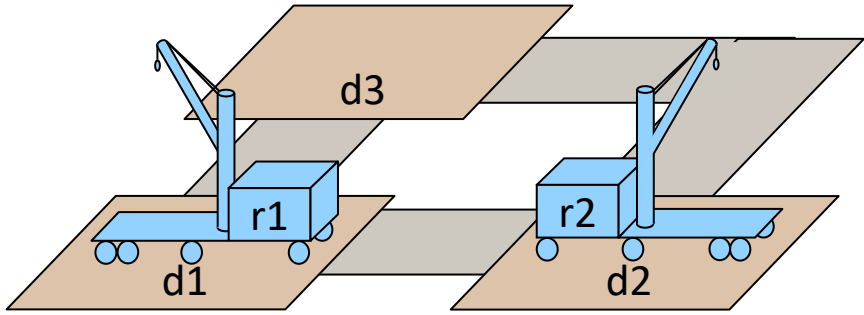
a_0 → a_g

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) ← d', occupied(d') = r, occupied(d) = nil

PSP Algorithm

```
PSP(Σ, π)
  loop
    if Flaws(π) = ∅ then return π
    arbitrarily select f ∈ Flaws(π)
    R ← {all feasible resolvers for f}
    if R = ∅ then return failure
    nondeterministically choose ρ ∈ R
    π ← ρ(π)
  return π
```



- 3 open goals
- no threats

a_0

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

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

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

a_g

select

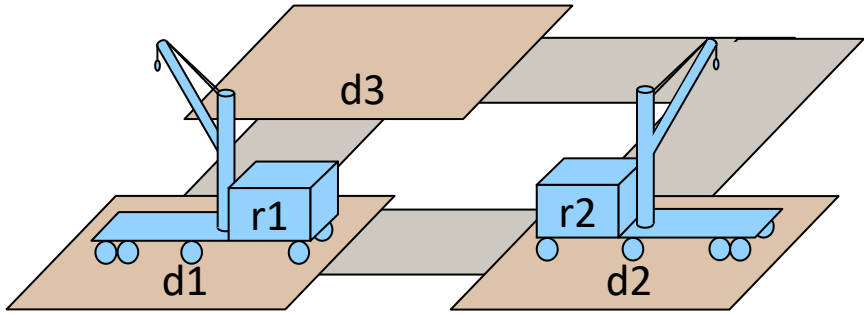
for every action a,

$a_0 \prec a \prec a_g$

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

PSP Algorithm

```
PSP(Σ, π)
  loop
    if Flaws(π) = ∅ then return π
    arbitrarily select f ∈ Flaws(π)
    R ← {all feasible resolvers for f}
    if R = ∅ then return failure
    nondeterministically choose ρ ∈ R
    π ← ρ(π)
  return π
```



- 4 open goals
- no threats

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

select →
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

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

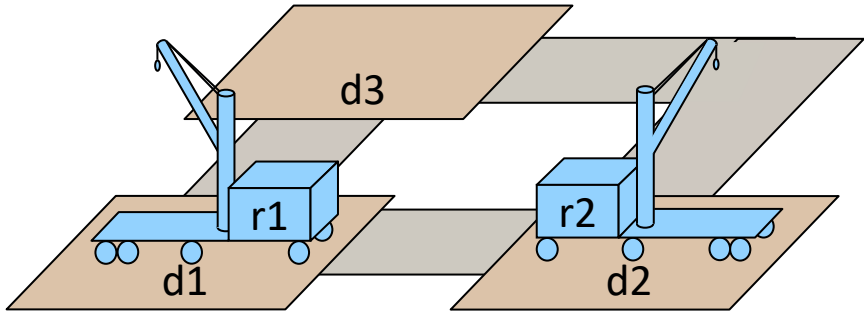
a_g

occupied(d1) = nil
loc(r2) = d'
a_2 = move(r2,d',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(π) = ∅ then return π
    arbitrarily select f ∈ Flaws(π)
    R ← {all feasible resolvers for f}
    if R = ∅ then return failure
    nondeterministically choose ρ ∈ R
    π ← ρ(π)
  return π
```



- 5 open goals
- 1 threat

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

only resolver: causal link from a new action

select

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

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

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

a_3 = move(r,d2,d'')
loc(r) = d2
occupied(d'') = nil
loc(r) = d''
occupied(d2) = nil
occupied(d'') = r

a_2 = move(r2,d',d1)
occupied(d1) = nil
loc(r2) = d'
loc(r2) = d1
occupied(d') = nil
occupied(d1) = r2

threat

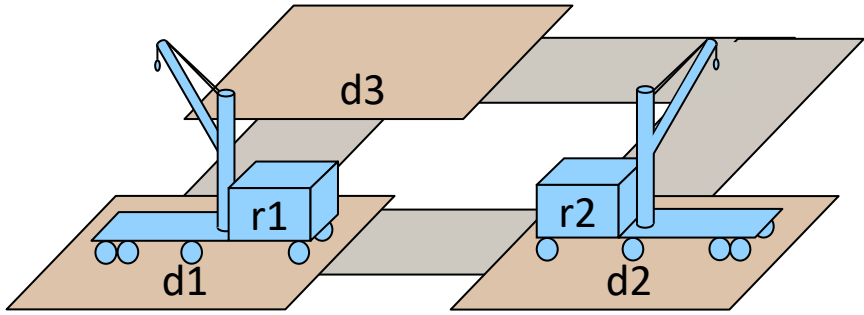
loc(r1) = d2
loc(r2) = d1

a_g

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

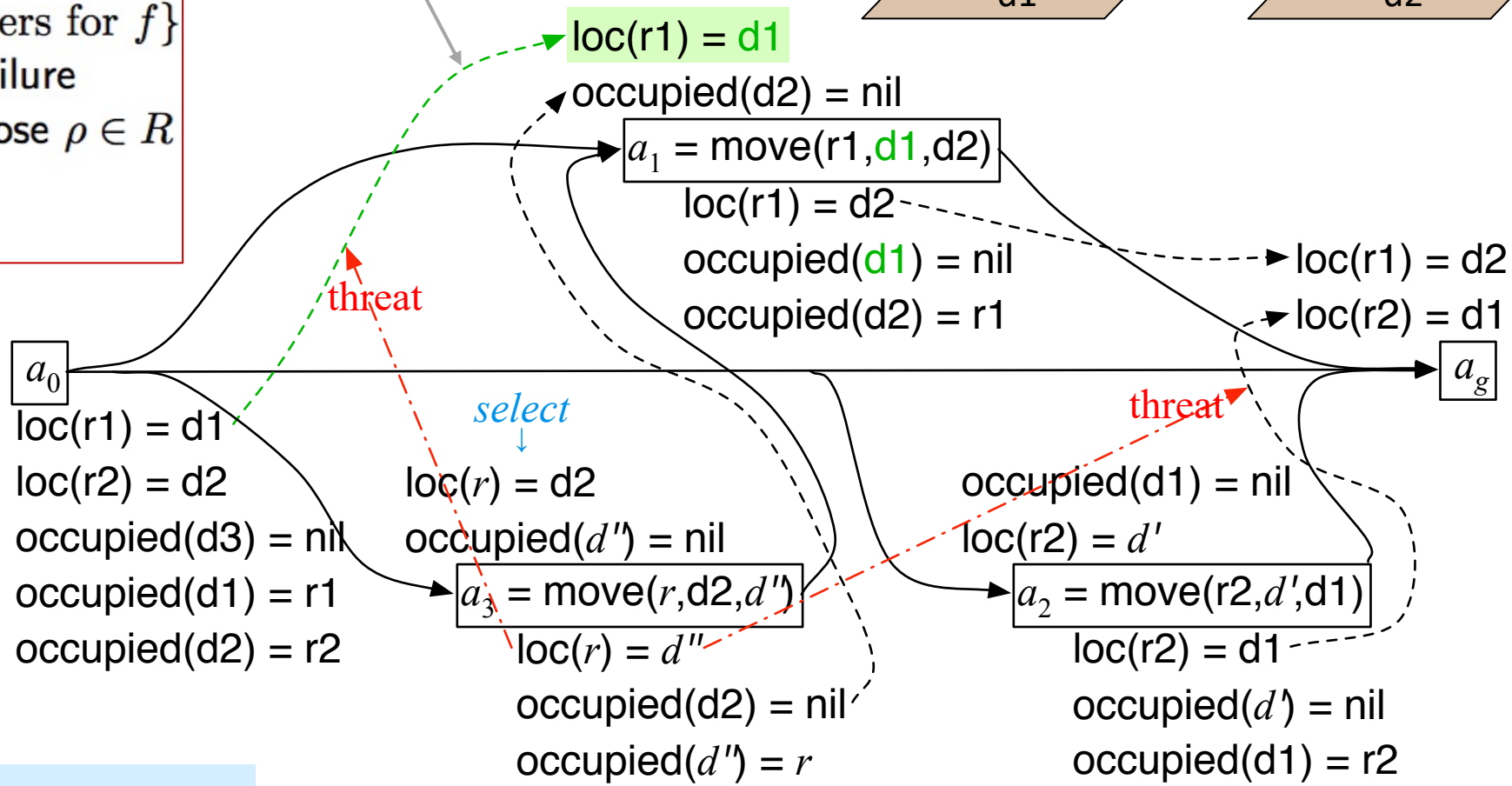
PSP Algorithm

```
PSP(Σ, π)
  loop
    if Flaws(π) = ∅ then return π
    arbitrarily select f ∈ Flaws(π)
    R ← {all feasible resolvers for f}
    if R = ∅ then return failure
    nondeterministically choose ρ ∈ R
    π ← ρ(π)
  return π
```



- 4 open goals
- 2 threats

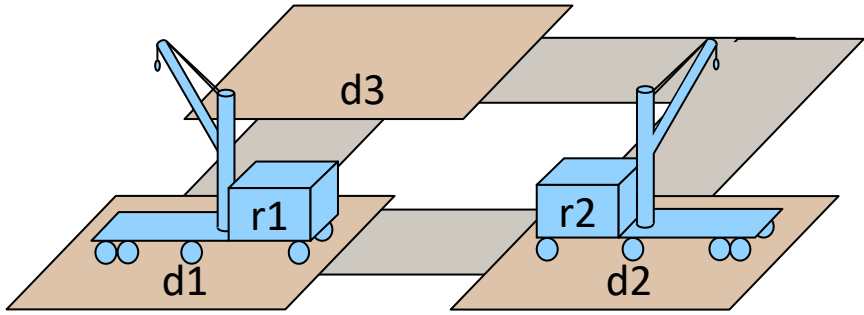
Poll: does a_3 threaten the causal link for a_g's precondition loc(r1)=d2?



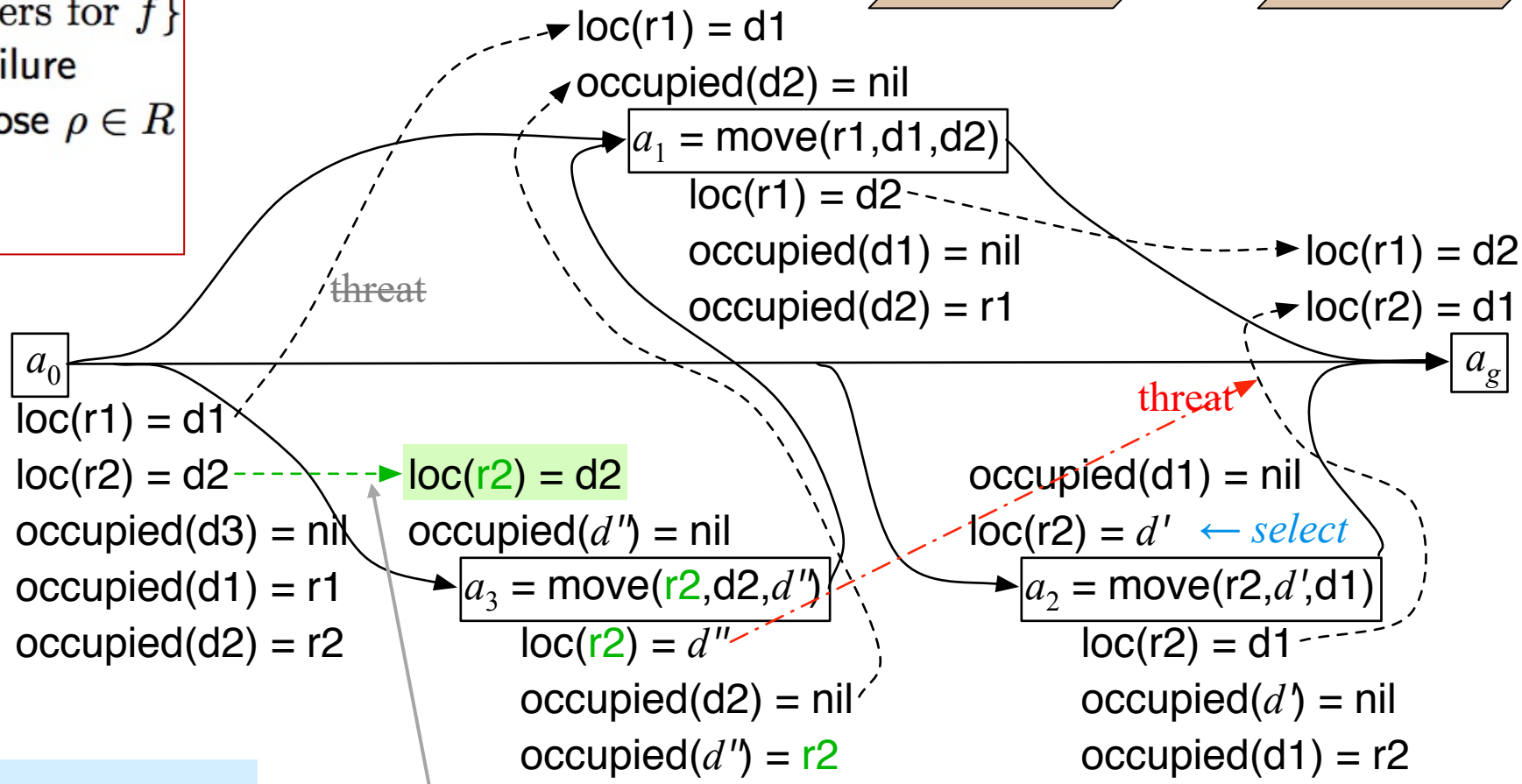
move(r, d, d')
pre: loc(r) = d, occupied(d') = nil
eff: loc(r) ← d', occupied(d') = r, occupied(d) = nil

PSP Algorithm

```
PSP(Σ, π)
  loop
    if Flaws(π) = ∅ then return π
    arbitrarily select f ∈ Flaws(π)
    R ← {all feasible resolvers for f}
    if R = ∅ then return failure
    nondeterministically choose ρ ∈ R
    π ← ρ(π)
  return π
```



- 3 open goals
- 1 threat

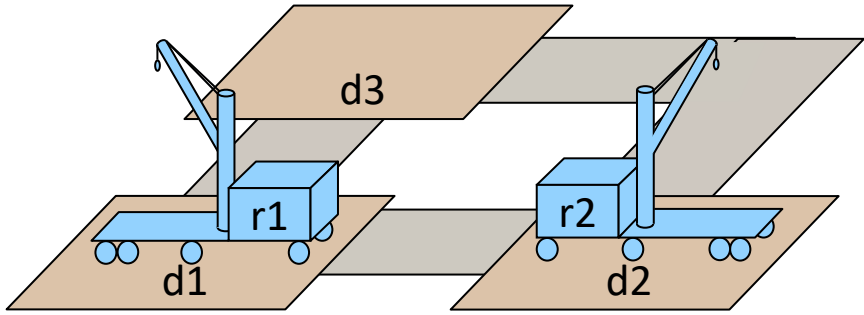


move(r, d, d')
pre: loc(r) = d, occupied(d') = nil
eff: loc(r) ← d', occupied(d') = r, occupied(d) = nil

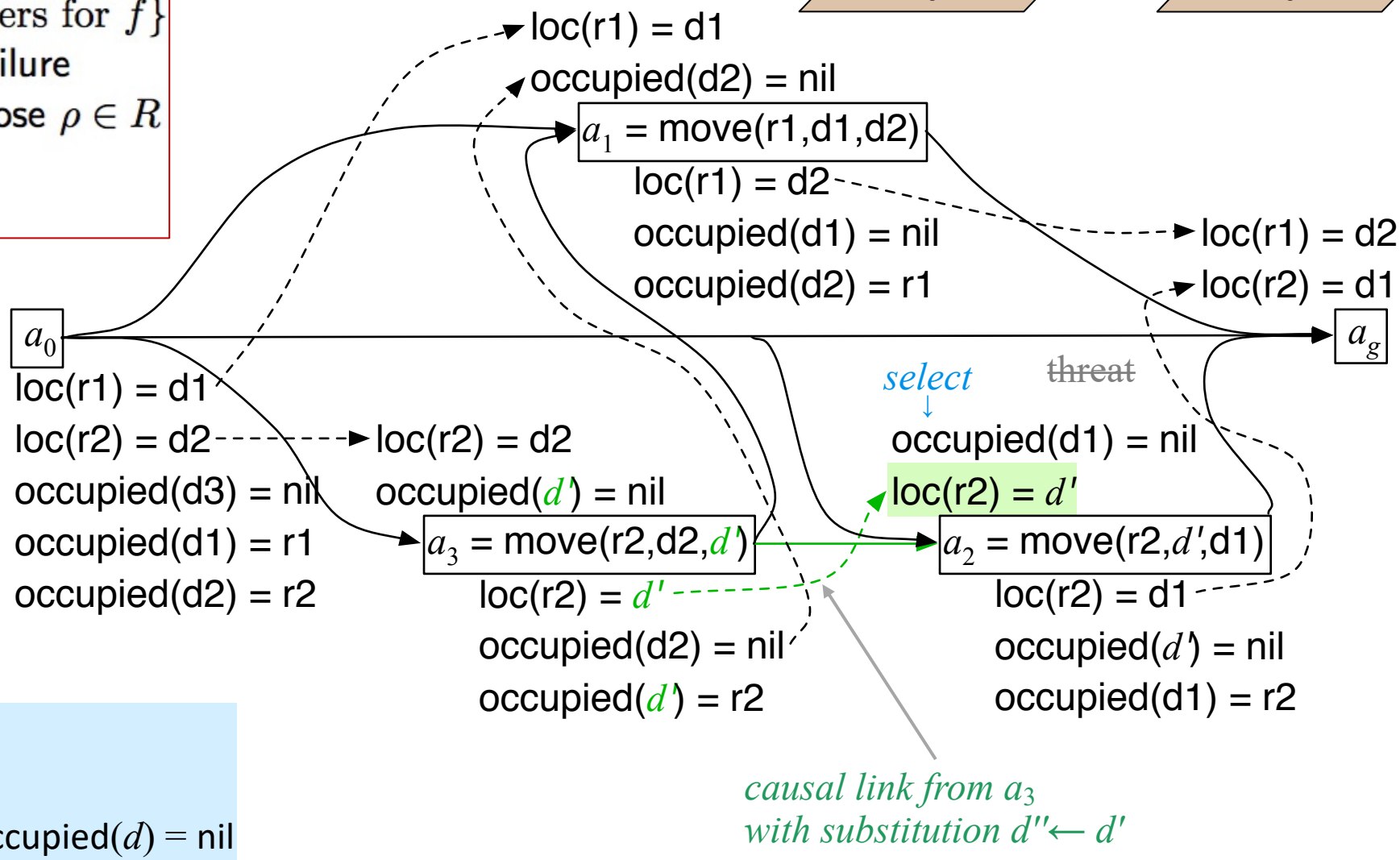
causal link from a_0 with substitution r←r2

PSP Algorithm

```
PSP(Σ, π)
  loop
    if Flaws(π) = ∅ then return π
    arbitrarily select f ∈ Flaws(π)
    R ← {all feasible resolvers for f}
    if R = ∅ then return failure
    nondeterministically choose ρ ∈ R
    π ← ρ(π)
  return π
```



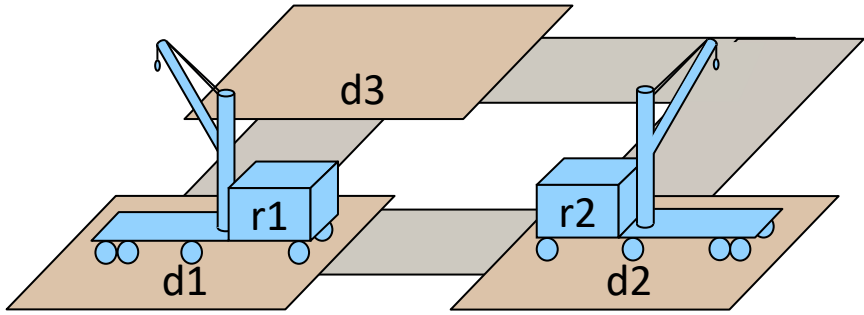
- 2 open goals
- no threats



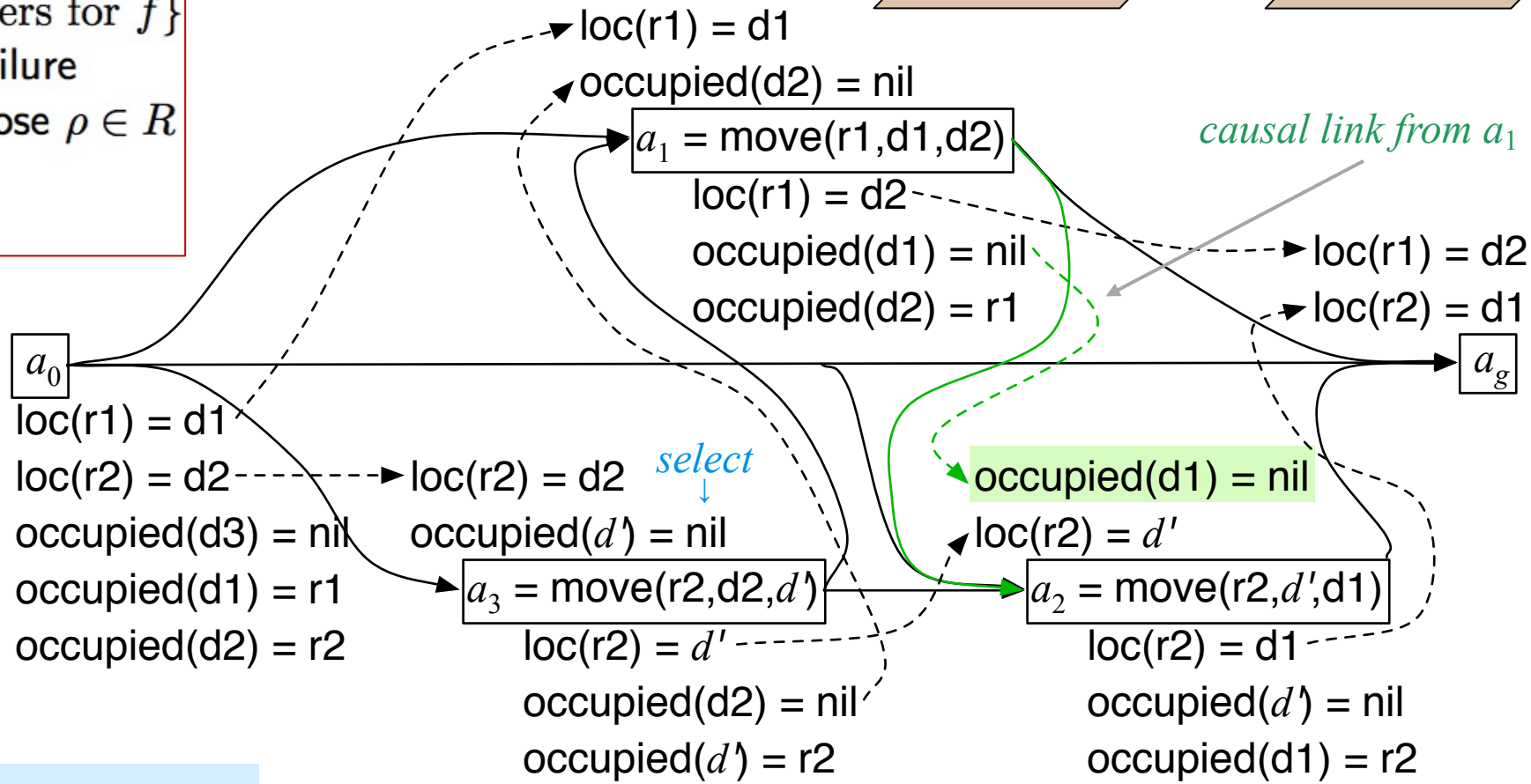
move(r, d, d')
pre: loc(r) = d, occupied(d') = nil
eff: loc(r) ← d', occupied(d') = r, occupied(d) = nil

PSP Algorithm

```
PSP(Σ, π)
  loop
    if Flaws(π) = ∅ then return π
    arbitrarily select f ∈ Flaws(π)
    R ← {all feasible resolvers for f}
    if R = ∅ then return failure
    nondeterministically choose ρ ∈ R
    π ← ρ(π)
  return π
```



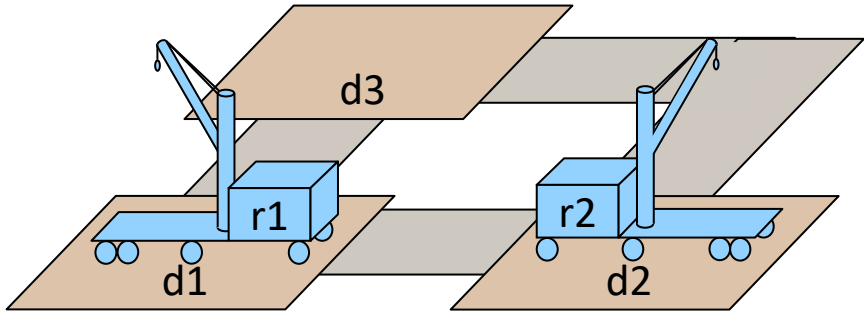
- 1 open goal
- no threats



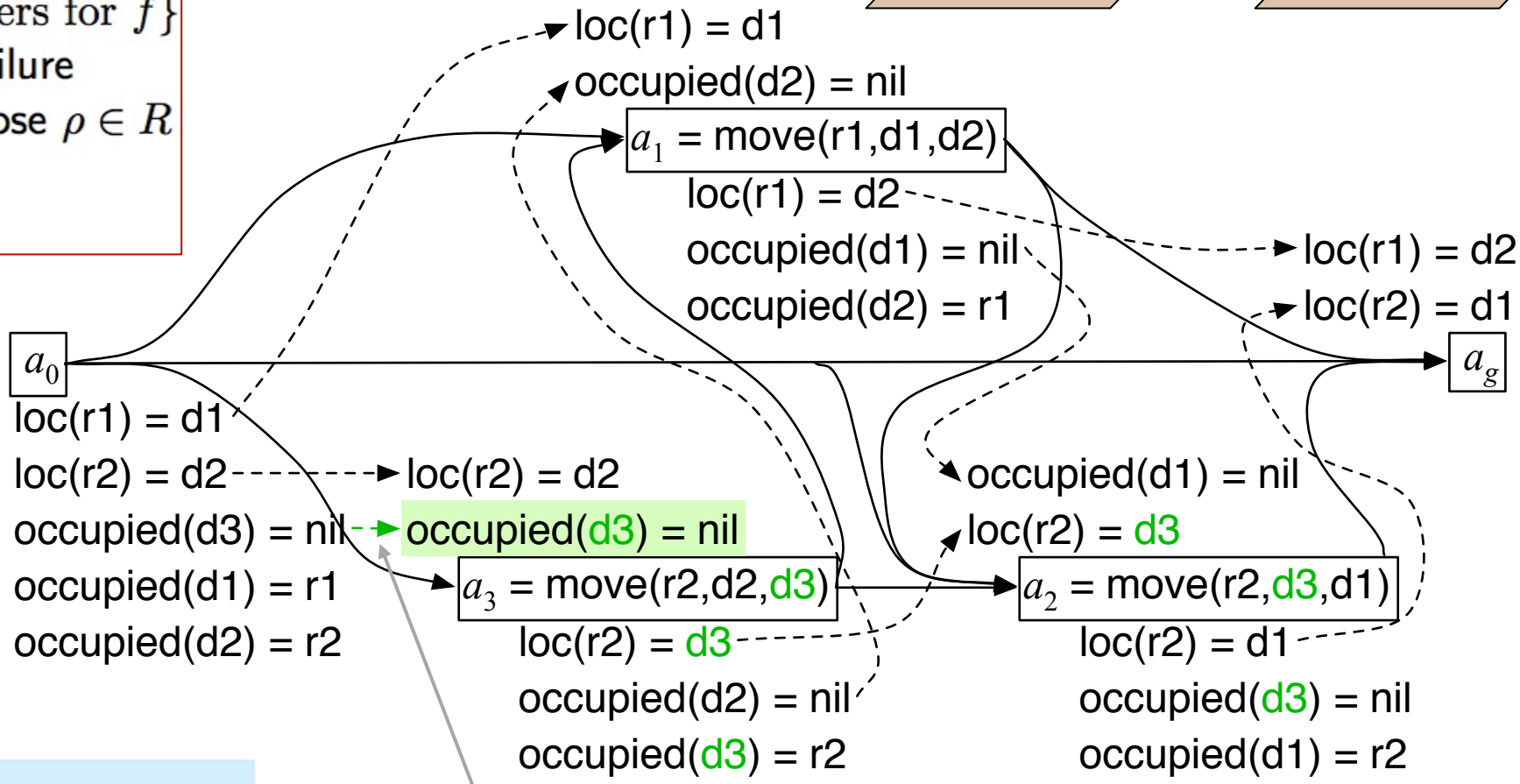
move(r, d, d')
pre: loc(r) = d, occupied(d') = nil
eff: loc(r) ← d', occupied(d') = r, occupied(d) = nil

PSP Algorithm

```
PSP(Σ, π)
  loop
    if Flaws(π) = ∅ then return π
    arbitrarily select f ∈ Flaws(π)
    R ← {all feasible resolvers for f}
    if R = ∅ then return failure
    nondeterministically choose ρ ∈ R
    π ← ρ(π)
  return π
```



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

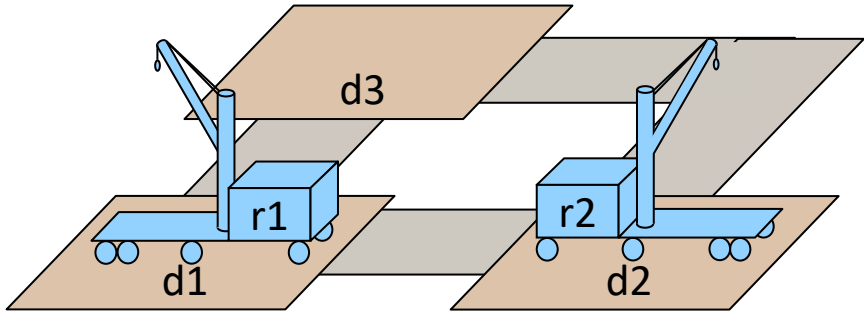


causal link from a_0 with substitution $d' \leftarrow$ d3

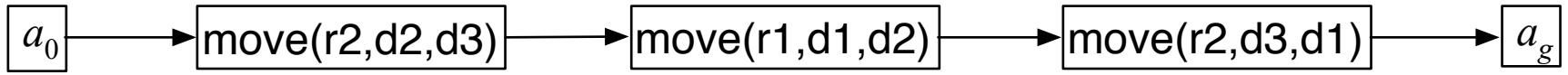
move(r, d, d')
pre: loc(r) = d, occupied(d') = nil
eff: loc(r) ← d', occupied(d') = r, occupied(d) = nil

PSP Algorithm

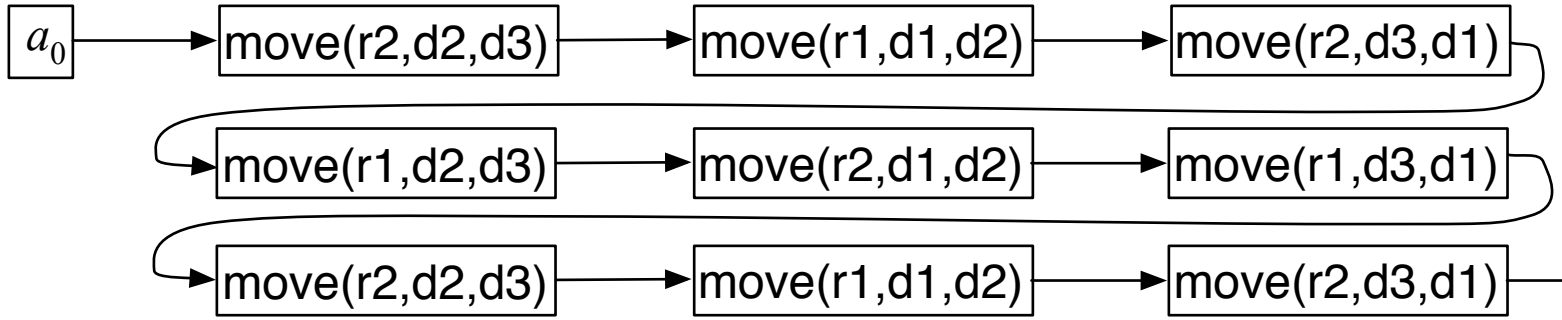
```
PSP(Σ, π)
  loop
    if Flaws(π) = ∅ then return π
    arbitrarily select f ∈ Flaws(π)
    R ← {all feasible resolvers for f}
    if R = ∅ then return failure
    nondeterministically choose ρ ∈ R
    π ← ρ(π)
  return π
```



- The solution we found:



- Another:

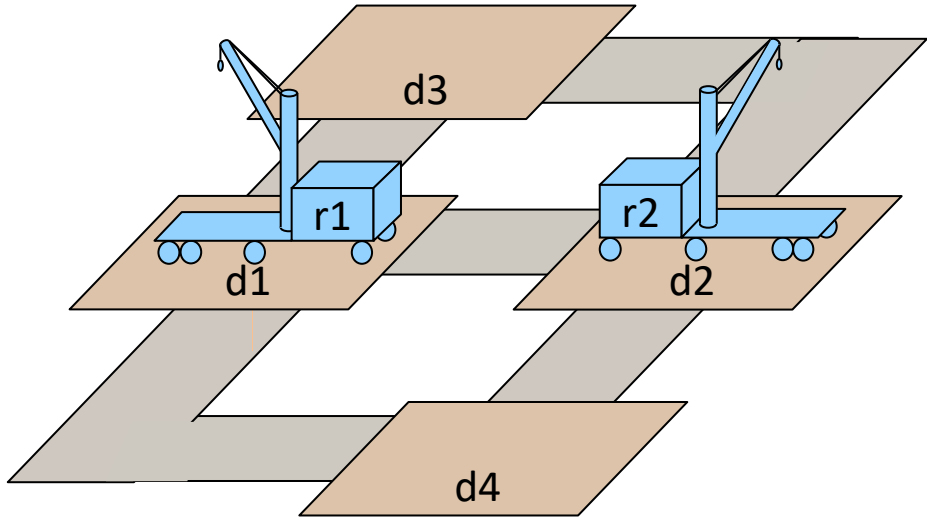


- Infinitely many others

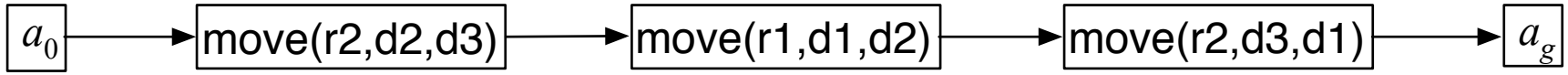
move(r, d, d')
pre: loc(r) = d, occupied(d') = nil
eff: loc(r) ← d', occupied(d') = r, occupied(d) = nil

PSP Algorithm

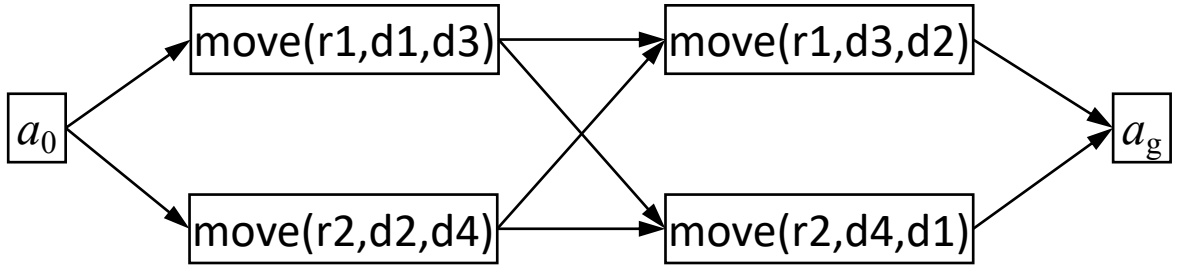
```
PSP(Σ, π)
  loop
    if Flaws(π) = ∅ then return π
    arbitrarily select f ∈ Flaws(π)
    R ← {all feasible resolvers for f}
    if R = ∅ then return failure
    nondeterministically choose ρ ∈ R
    π ← ρ(π)
  return π
```



- The totally-ordered solutions we found earlier



- And partially-ordered solutions

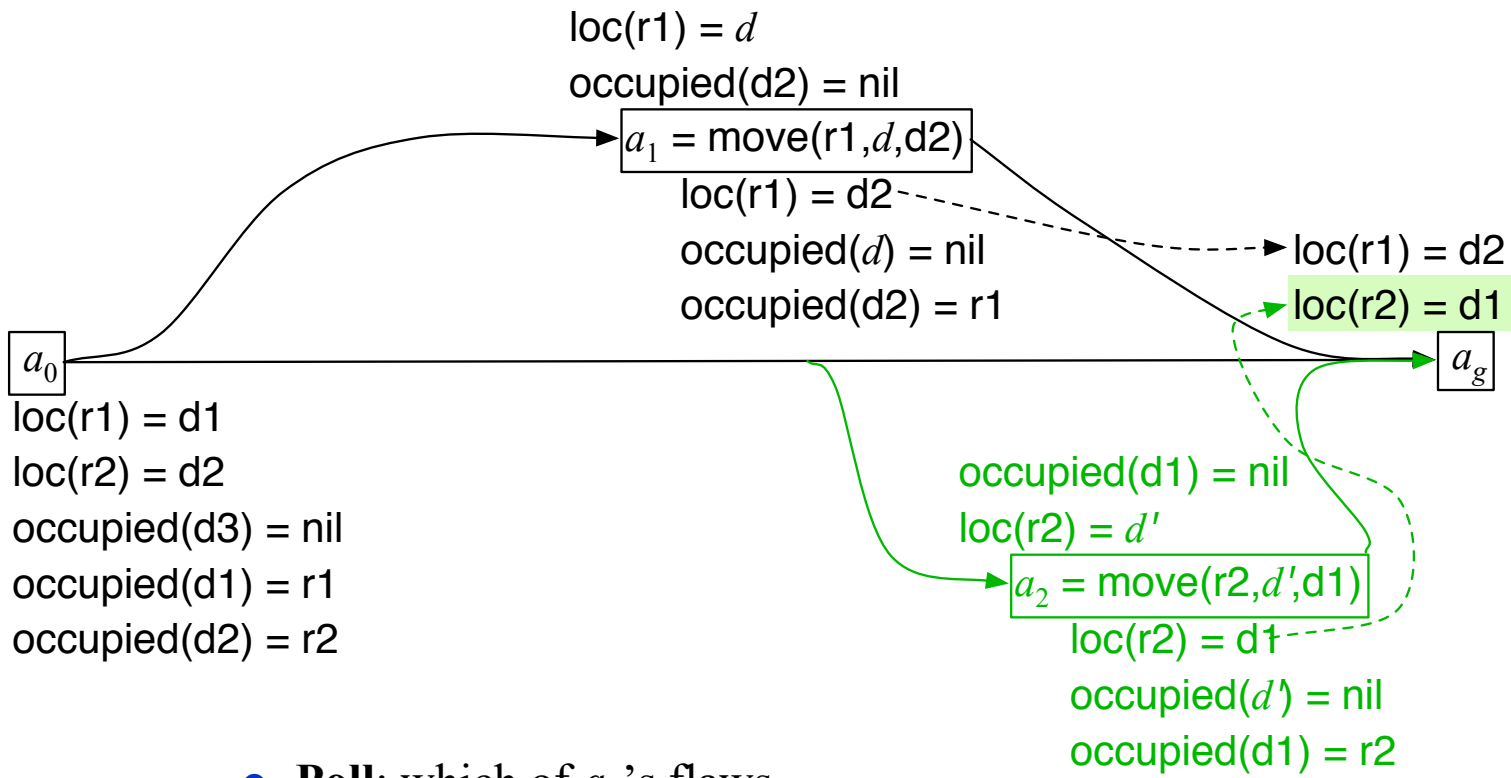


move(r, d, d')
pre: loc(r) = d, occupied(d') = nil
eff: loc(r) ← d', occupied(d') = r, occupied(d) = nil

Selecting a Flaw

```
PSP(Σ, π)
  loop
    if Flaws(π) = ∅ then return π
    arbitrarily select f ∈ Flaws(π)
    R ← {all feasible resolvers for f}
    if R = ∅ then return failure
    nondeterministically choose ρ ∈ R
    π ← ρ(π)
  return π
```

- Resolving a flaw in PSP
  ≈ assigning a value to a variable in a CSP
- *Fewest Alternatives First (FAF):*
  - select flaw with fewest resolvers
    ≈ Minimum Remaining Values (MRV) heuristic for CSPs



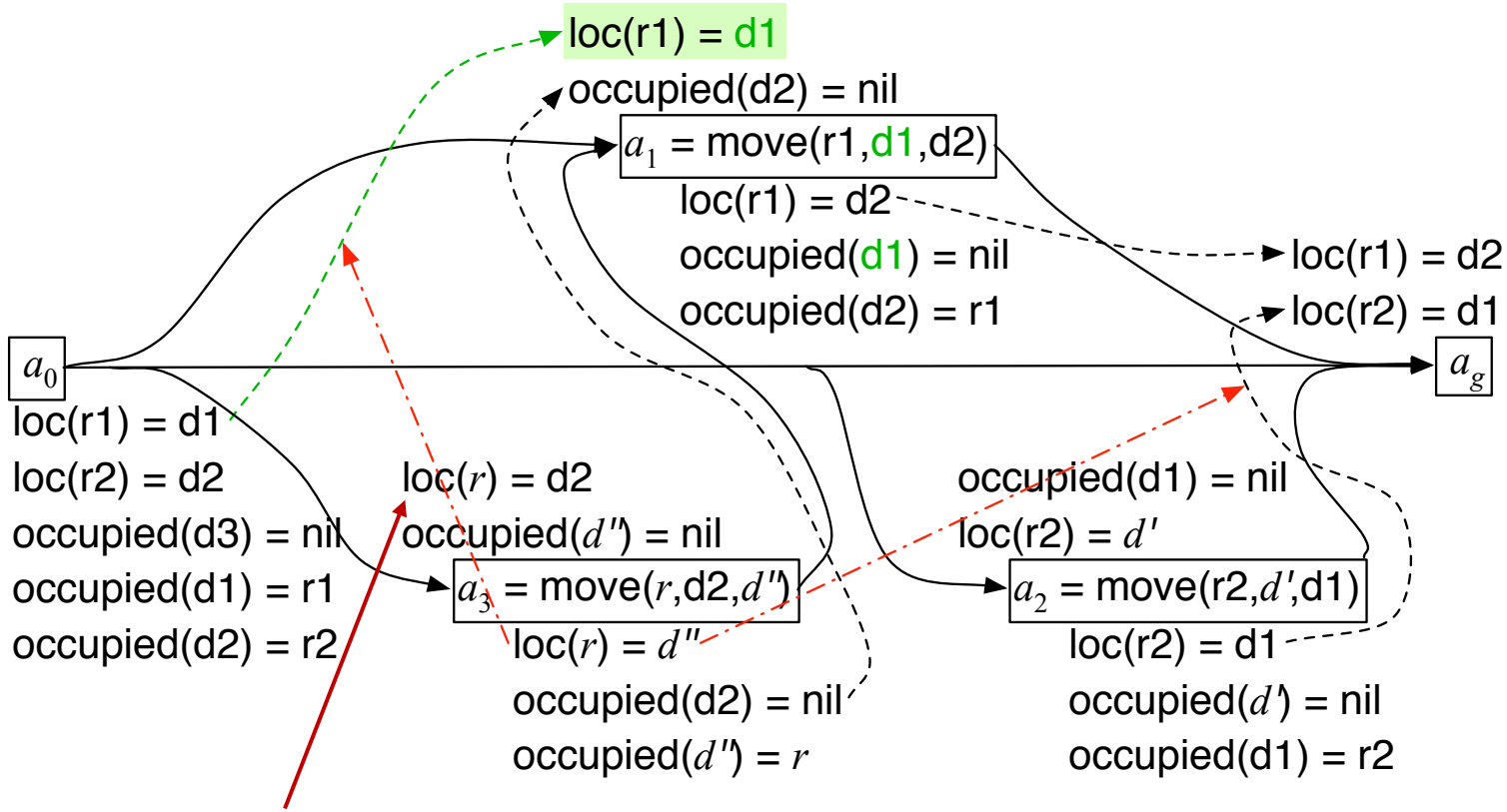
- **Poll**: which of a_1's flaws would FAF select first?
  1. loc(r1)=d
  2. occupied(d2) = nil
  3. no preference

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(π) = ∅ then return π
    arbitrarily select f ∈ Flaws(π)
    R ← {all feasible resolvers for f}
    if R = ∅ then return failure
    nondeterministically choose ρ ∈ R
    π ← ρ(π)
  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

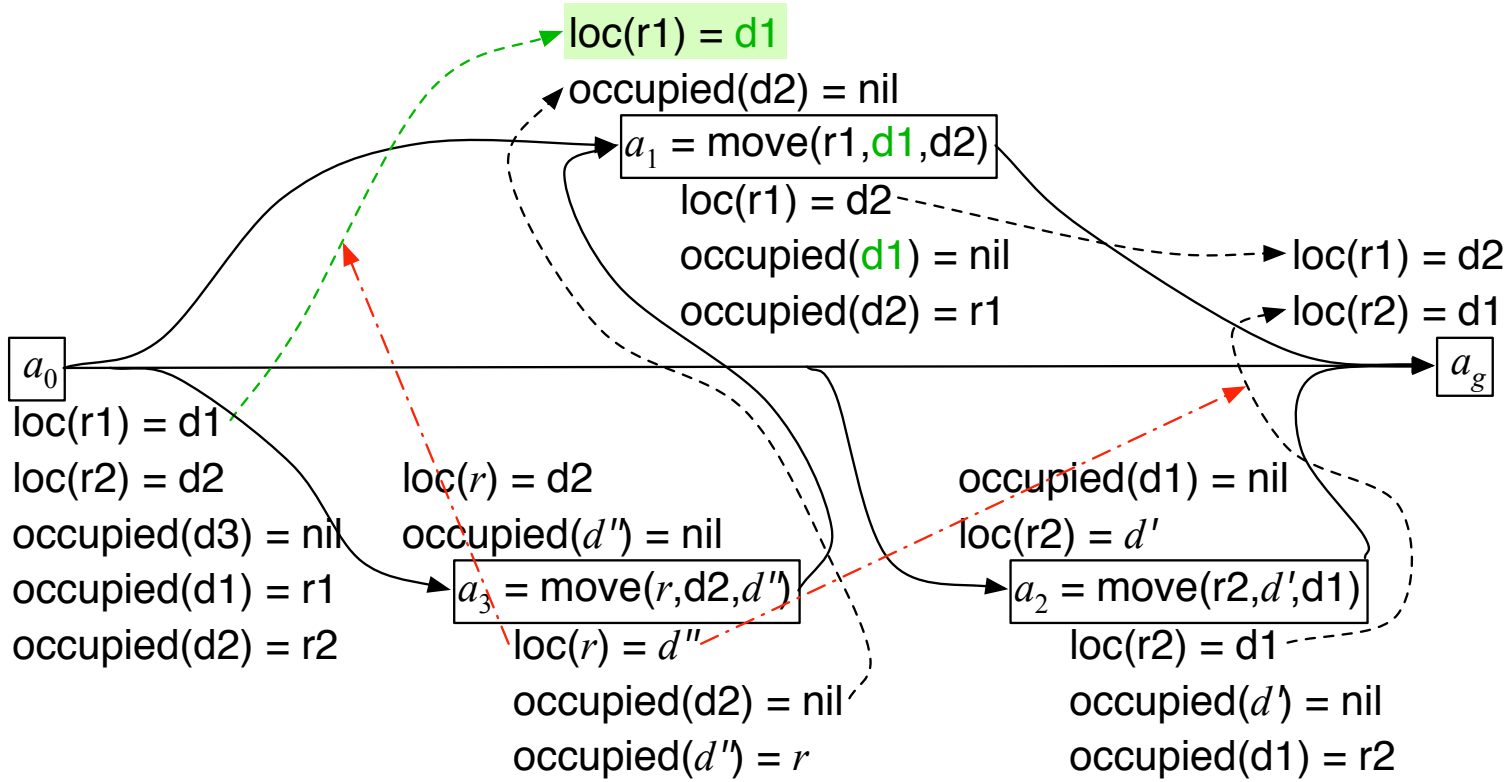


- **Poll**: for loc(r)=d2 in a_3, which resolver would LCR choose first?
  1. causal link from a new action
  2. causal link from a_0, with substitution r←r2
  3. no preference

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(π) = ∅ then return π
    arbitrarily select f ∈ Flaws(π)
    R ← {all feasible resolvers for f}
    if R = ∅ then return failure
    nondeterministically choose ρ ∈ R
    π ← ρ(π)
  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):
  - 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

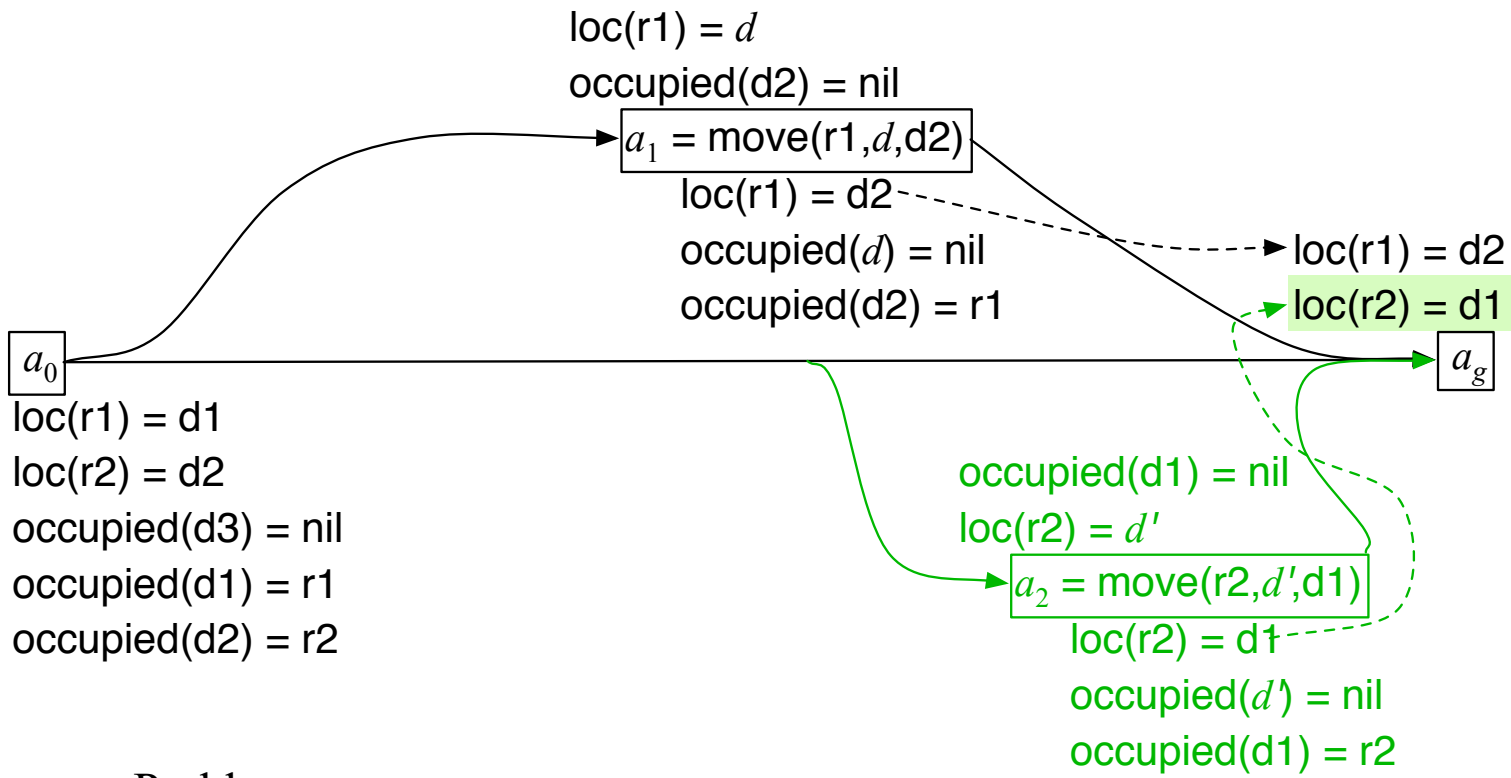
Choosing a Resolver

```
PSP(Σ, π)
  loop
    if Flaws(π) = ∅ then return π
    arbitrarily select f ∈ Flaws(π)
    R ← {all feasible resolvers for f}
    if R = ∅ then return failure
    nondeterministically choose ρ ∈ R
    π ← ρ(π)
  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):
  - 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:
  - For loc(r1)=d in a_1, ANA chooses a_0 with substitution d←d1
  - For loc(r2)=d' in a_2, ANA chooses a_0 with substitution d'←d2
  - Makes the problem unsolvable
- Perhaps use ANA anyway?

Discussion

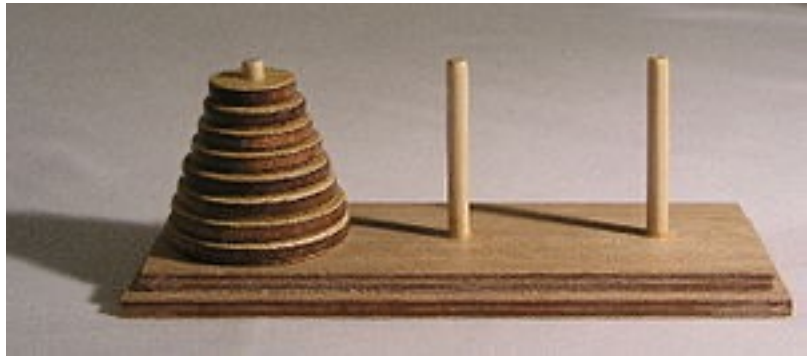
- Problem: how to prune infinitely long paths in the search space?
  - Loop detection is based on recognizing states or goals we've seen before
  - Partially ordered plan: don't know the states

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

- 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 of $|S|$ or more actions
  - Not very helpful

- Unclear whether there's a better pruning technique

Summary and Addenda

- 2.5 Plan-Space Search
  - Partially ordered plans and solutions
  - partial plans, causal links
  - flaws: open goals, threats, resolvers
  - PSP algorithm, long example, node-selection heuristics
