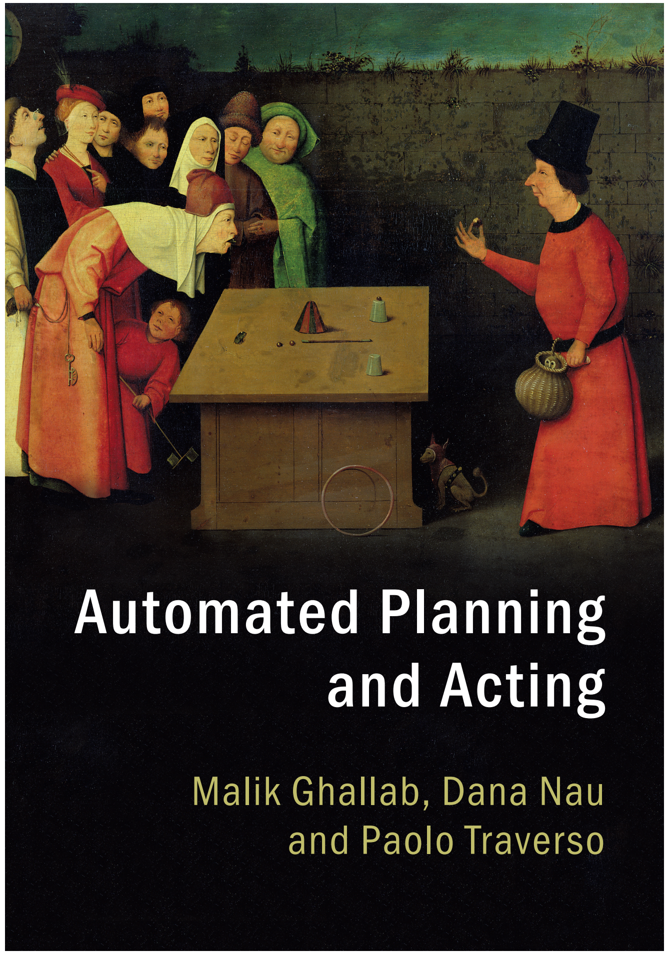
Last update: 3:25 PM, March 10, 2022

Chapter 2

Deliberation with Deterministic Models

2.3: Heuristic Functions
2.7.7: HTN Planning

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
- 2.7.7 HTN planning

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

- 2.4 Backward search
- 2.5 Plan-space search

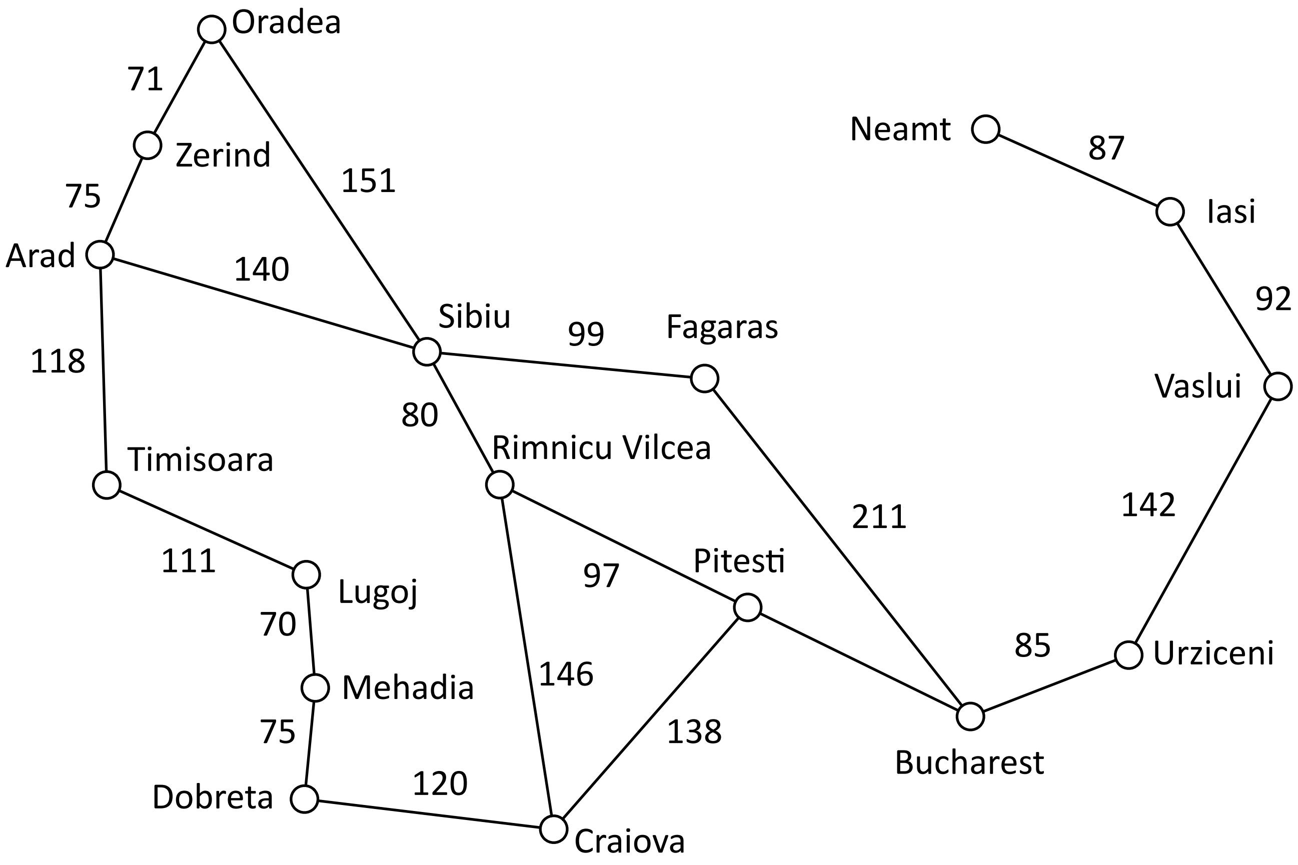
Additional slides:

- 2.7.8 LTL_planning.pdf

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

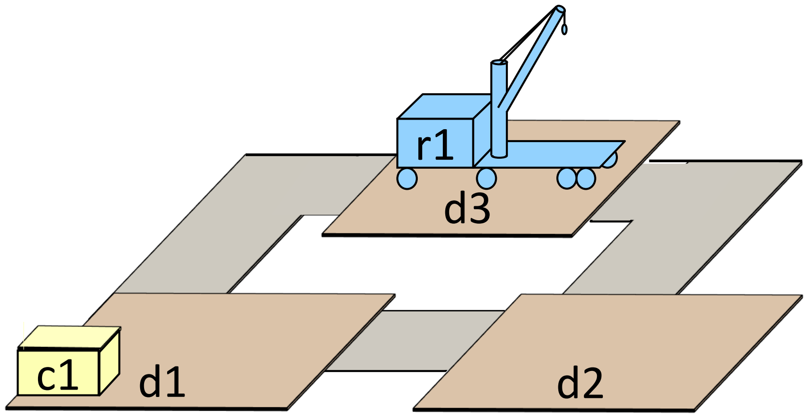
Domain-independent Heuristics

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

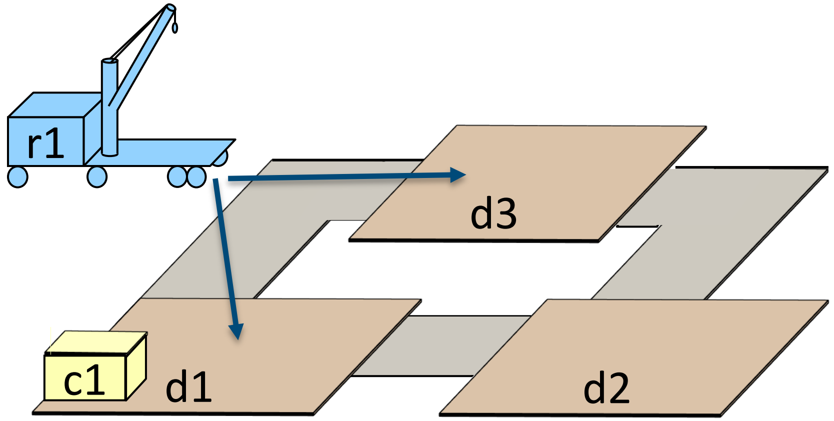
    In the book, but I'll skip them
  - Delete-relaxation heuristics
    - Optimal relaxed solution
    - Fast-forward heuristic
  - Landmark heuristics

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}

$\hat{s}_1 = \gamma^+(s_0$, 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

A. Yes B. No C. don't know

Relaxed Applicability

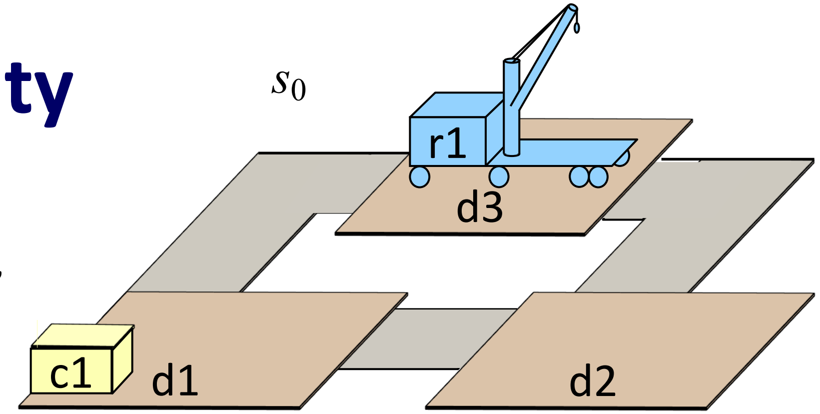
- Action a is *r-applicable* in a relaxed state $\hat{s}$ if an *r-subset* of $\hat{s}$ satisfies a's preconditions
  - a subset with one value per state variable
- If a is r-applicable then $\gamma^+(\hat{s},a) = \hat{s} \cup \gamma(s,a)$

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

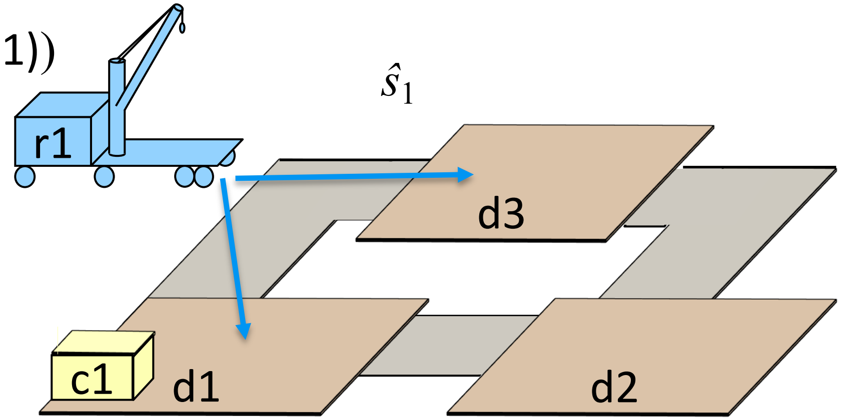
move(r, d, e)
  pre: loc(r)=d
  eff: loc(r)←e

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

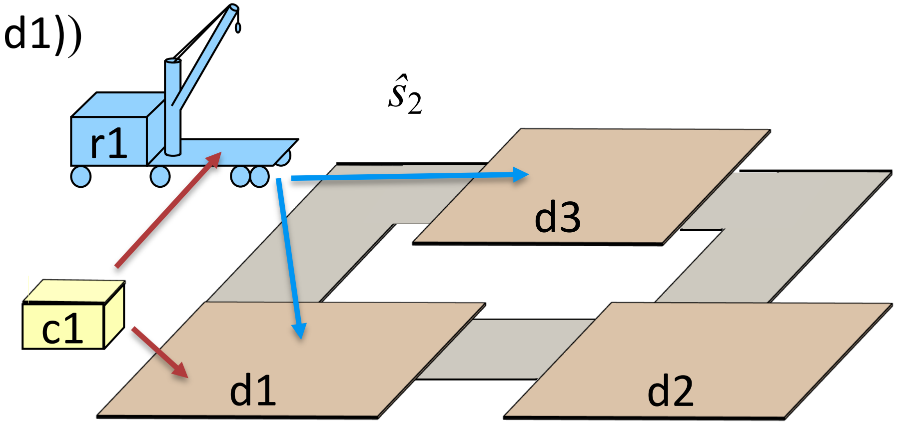
s_0 = {loc(r1) = d3,
cargo(r1) = nil,
loc(c1) = d1}



$\hat{s}_1 = \gamma^+(s_0,$ 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}

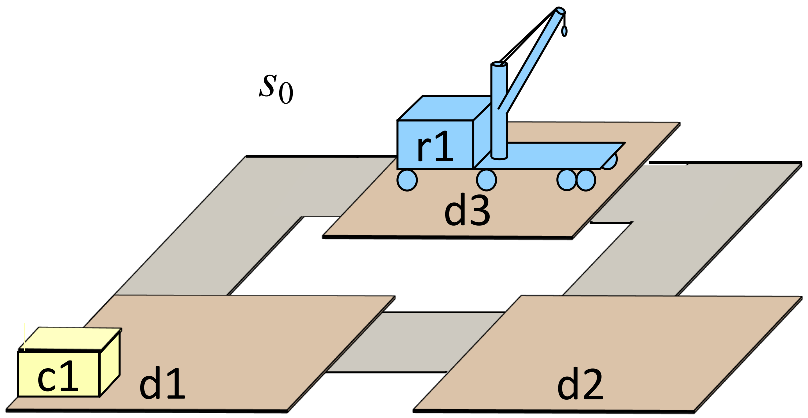


Relaxed Applicability (continued)

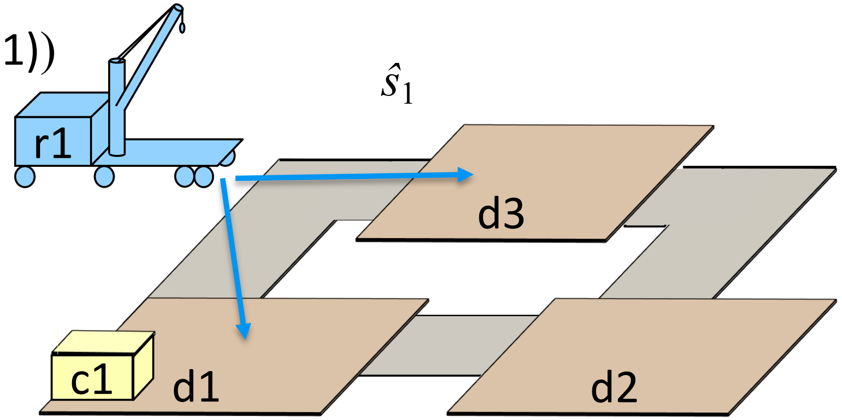
- Let $\pi = \langle a_1, \ldots, a_n \rangle$ be a plan
- Suppose we can r-apply the actions of π in the order $a_1, \ldots, a_n$:
  - r-apply a_1 in $\hat{s}_0$, get $\hat{s}_1 = \gamma^+(\hat{s}_0, a_1)$
  - r-apply a_2 in $\hat{s}_1$, get $\hat{s}_2 = \gamma^+(\hat{s}_1, a_2)$
  - …
  - r-apply a_n 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

  $\gamma^+(s_0,$ ⟨move(r1,d3,d1), load(r1,c1,d1)⟩$) = \hat{s}_2$

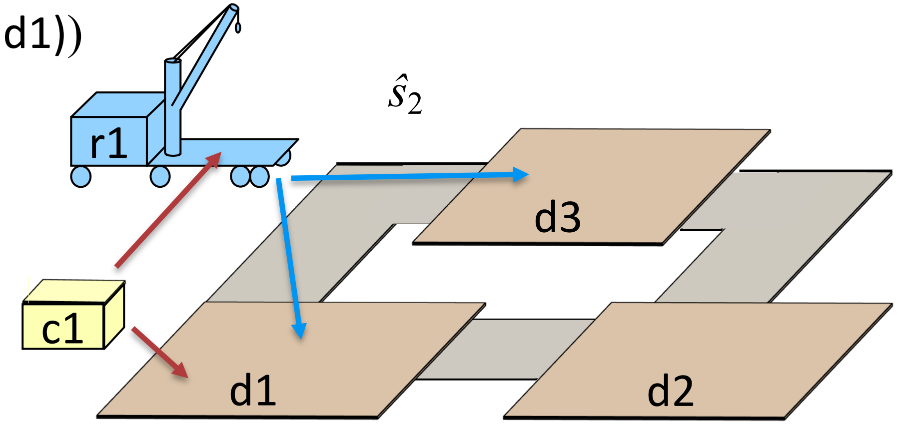
$s_0 =$ {loc(r1) = d3,
cargo(r1) = nil,
loc(c1) = d1}



$\hat{s}_1 = \gamma^+(s_0,$ 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}



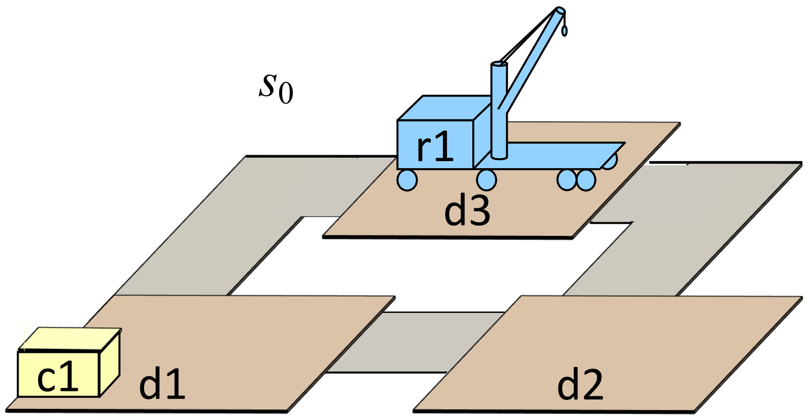
Relaxed Solution

- An r-state $\hat{s}$ *r-satisfies* a formula g if an r-subset of $\hat{s}$ satisfies g

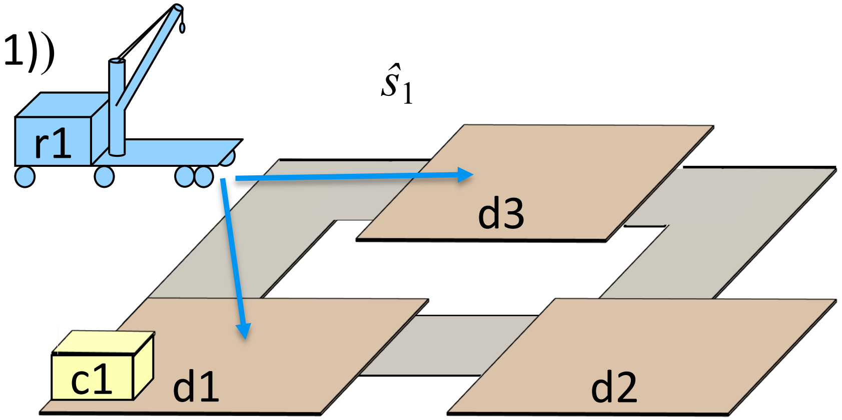
- *Relaxed solution* for a planning problem $P = (\Sigma, s_0, g)$:
  - a plan π such that $\gamma^+(s_0, \pi)$ r-satisfies g

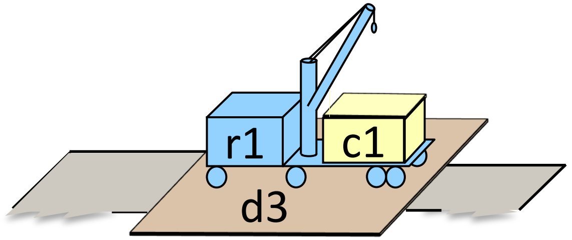
- Example: let P be as shown
  - $\hat{s}_2$ r-satisfies g
  - So $\pi = \langle$move(r1,d3,d1), load(r1,c1,d1)$\rangle$ is a relaxed solution for P

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



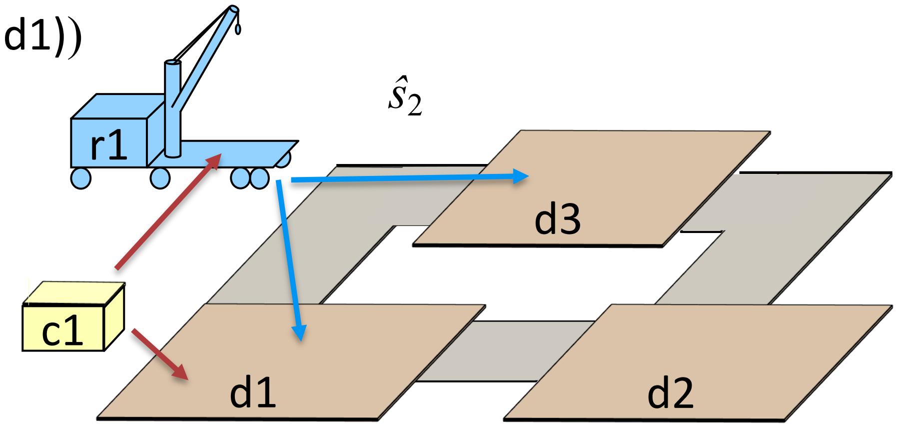
$\hat{s}_1 = \gamma^+(s_0,$ move(r1,d3,d1))
= {loc(r1) = d1,
loc(r1) = d3,
cargo(r1) = nil,
loc(c1) = d1}





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

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

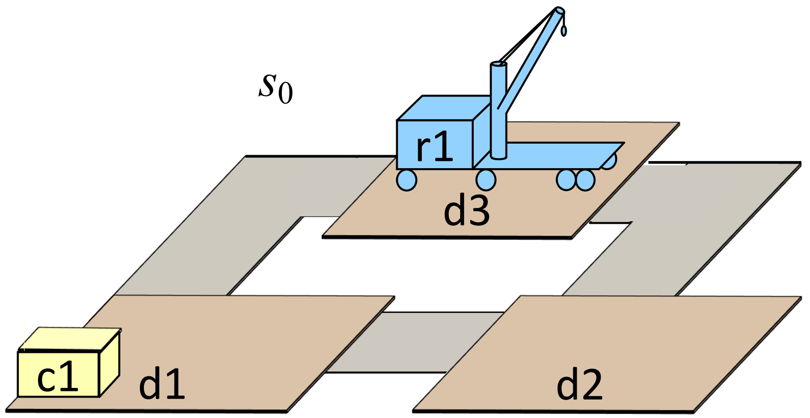


Relaxed Solution

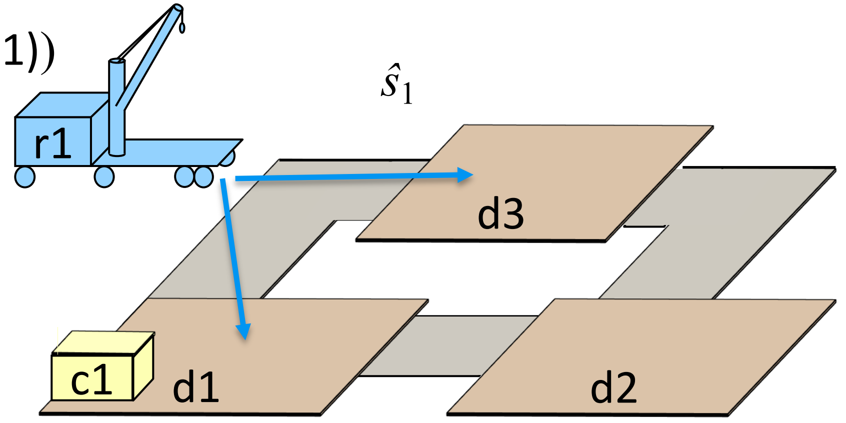
- Planning problem $P = (\Sigma, s_0, g)$
- *Optimal relaxed solution* heuristic:
  - $h^+(s)$ = minimum cost of all relaxed solutions for (Σ, s, g)

- Example: $s = s_0$
- $\pi = \langle$move(r1,d3,d1), load(r1,c1,d1)$\rangle$
  - cost(π) = 2
- No less-costly relaxed solution, so $h^+(s_0) = 2$

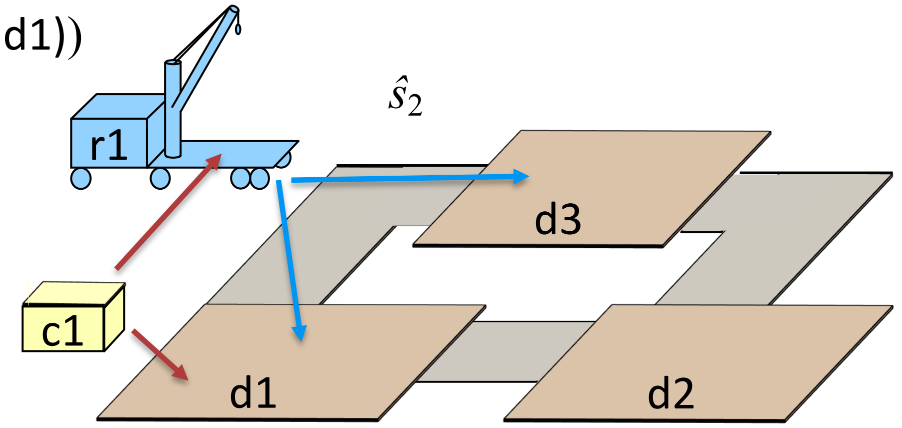
s_0 = {loc(r1) = d3,
cargo(r1) = nil,
loc(c1) = d1}



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



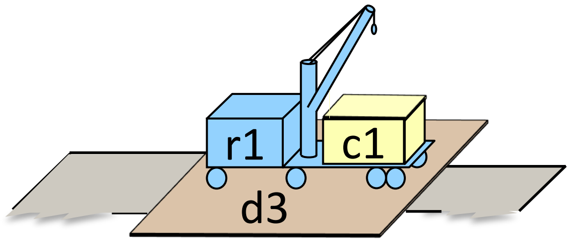
$\hat{s}_2 = \gamma^+(\hat{s}_1,$ load(r1,c1,d1))
= {loc(r1) = d1,
loc(r1) = d3,
cargo(r1) = nil,
cargo(r1) = c1,
loc(c1) = r1,
loc(c1) = d1}



Poll: is h^+ admissible?

A. Yes

B. No



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

Example: GBFS

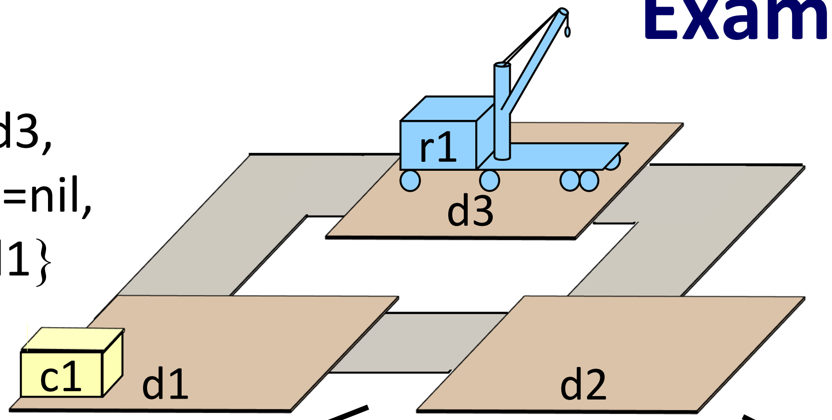
Poll 1: What is $h^+(s_1)$?

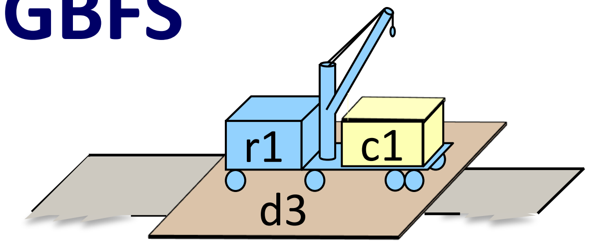
A. 1
B. 2
C. 3
D. 4
E. other

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

A. 1
B. 2
C. 3
D. 4
E. other

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



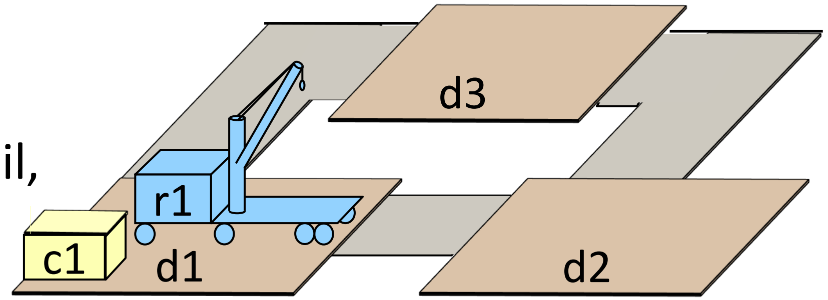


g = {loc(r1)=d3, loc(c1)=r1}

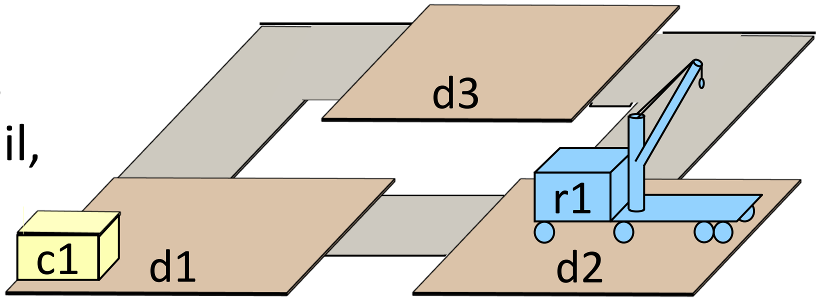
a_1 = move(r1,d3,d1)

a_2 = move(r1,d3,d2)

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



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



- GBFS with initial state s_0, goal g, heuristic h^+
- Applicable actions a_1, a_2 produce states s_1, s_2
- GBFS computes $h^+(s_1)$ and $h^+(s_2)$, chooses the state that has the lower h^+ value

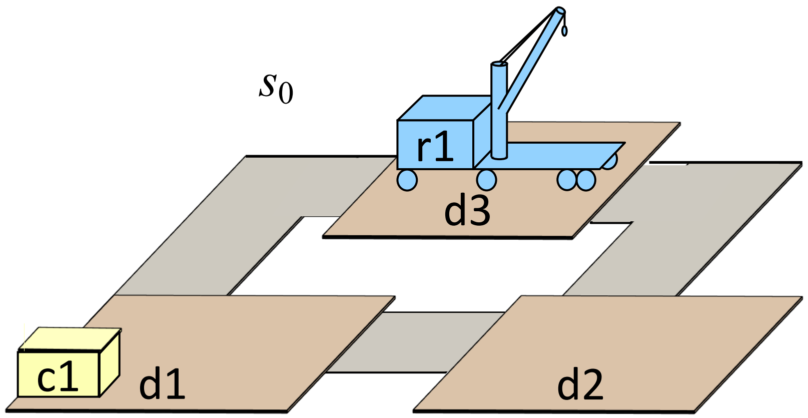
Fast-Forward Heuristic

- 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*

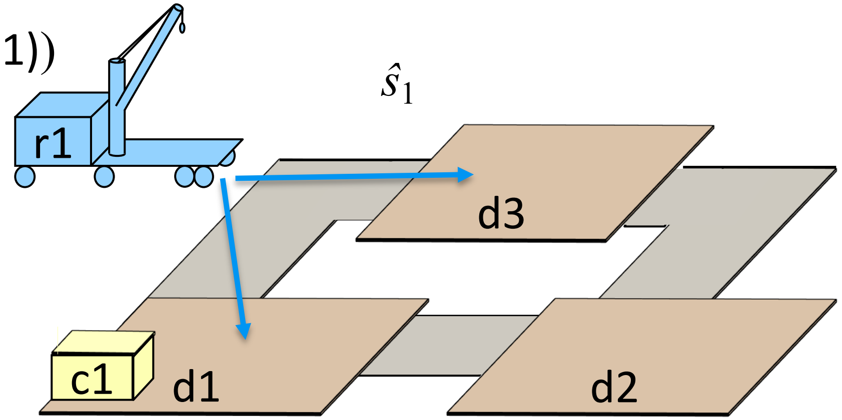
Relaxed Solution



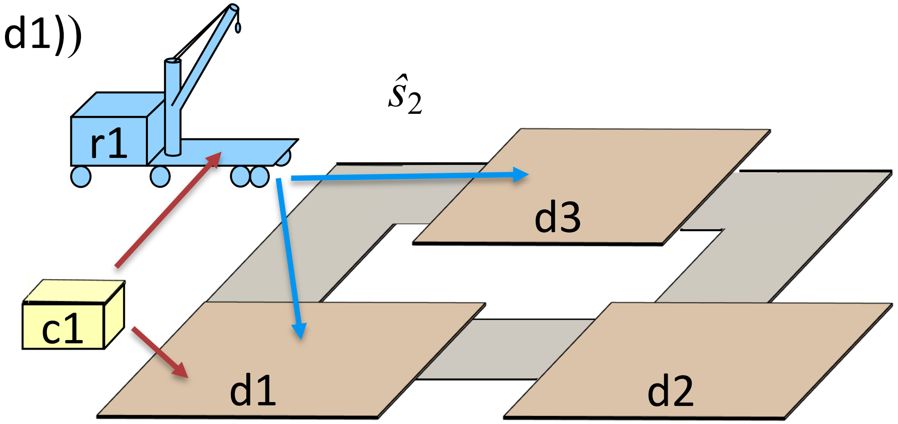
s_0 = {loc(r1) = d3,
cargo(r1) = nil,
loc(c1) = d1}

- Planning problem $P = (\Sigma, s_0, g)$
- *Optimal relaxed solution* heuristic:
  - $h^+(s)$ = minimum cost of all relaxed solutions for (Σ, s, g)
- Example: $s = s_0$
- $\pi = \langle$move(r1,d3,d1), load(r1,c1,d1)$\rangle$
  - cost(π) = 2
- No less-costly relaxed solution, so $h^+(s_0) = 2$

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



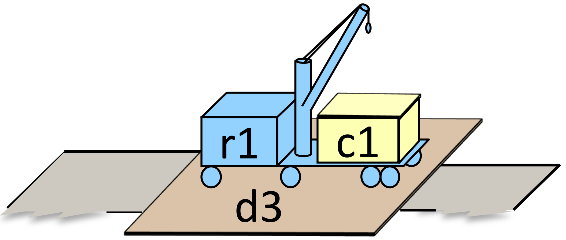
$\hat{s}_2 = \gamma^+(\hat{s}_1,$ load(r1,c1,d1))
= {loc(r1) = d1,
loc(r1) = d3,
cargo(r1) = nil,
cargo(r1) = c1,
loc(c1) = r1,
loc(c1) = d1}



Poll: is h^+ admissible?

A. Yes

B. No



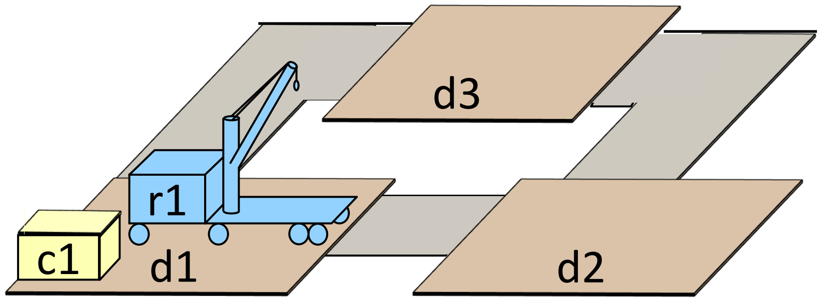
g = {loc(r1)=d3, loc(c1)=r1}

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_1 in the opposite order $\Rightarrow$ 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



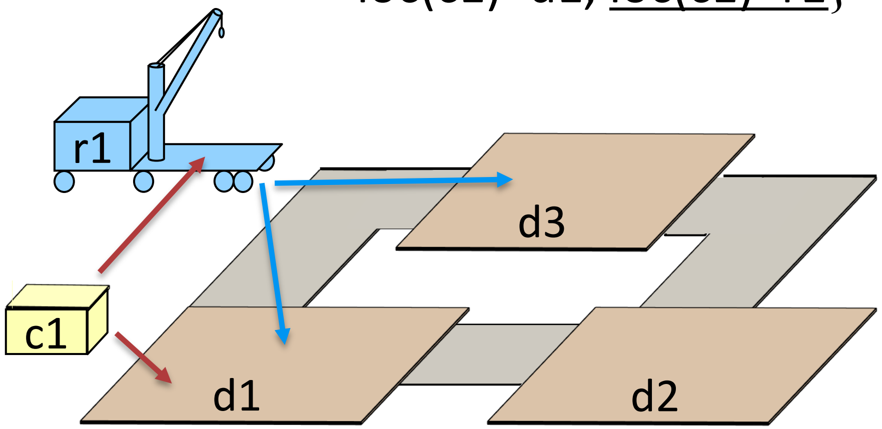
s_0 = {loc(r1)=d1, cargo(r1)=nil, loc(c1)=d1}

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, <u>loc(r1)=d3</u>,
cargo(r1)=nil, <u>cargo(r1)=c1</u>,
loc(c1)=d1, <u>loc(c1)=r1</u>}



Fast-Forward Heuristic

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 $\hat{s}_k$ 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^+

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

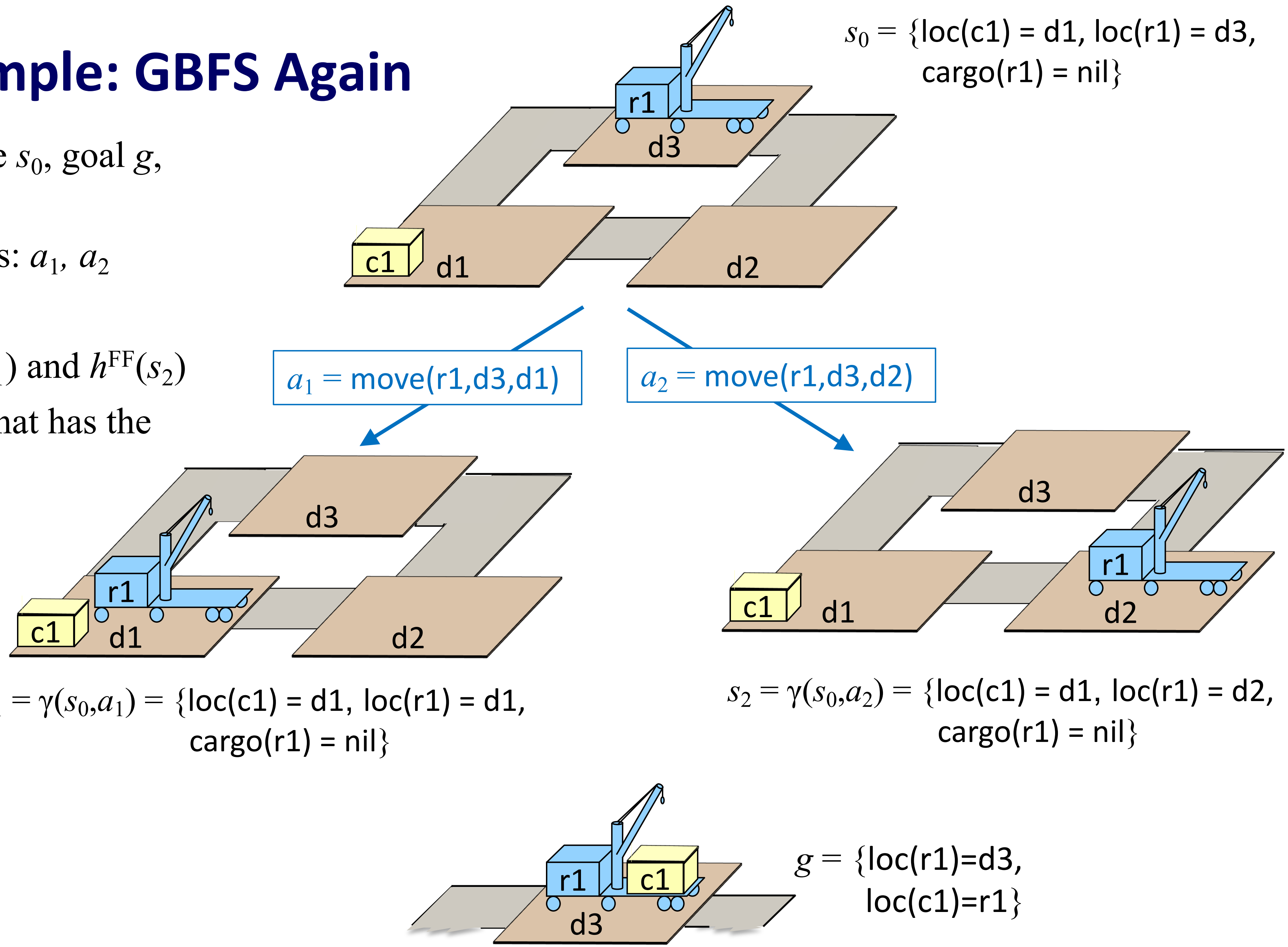
$\text{eff}(\hat{a}_i) = \bigcup \{\text{eff}(a) \mid a \in \hat{a}_i\}$

ambiguous →

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

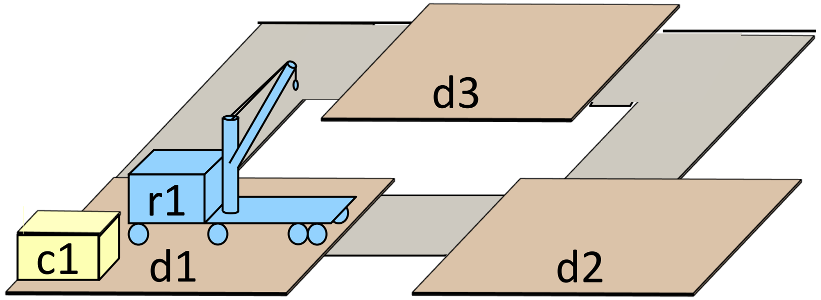
Example: GBFS Again

- 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 $h^{\mathrm{FF}}(s_1)$ and $h^{\mathrm{FF}}(s_2)$
  - Chooses the state that has the lower h^{FF} value
- Next several slides:
  - $h^{\mathrm{FF}}(s_1)$
  - $h^{\mathrm{FF}}(s_2)$

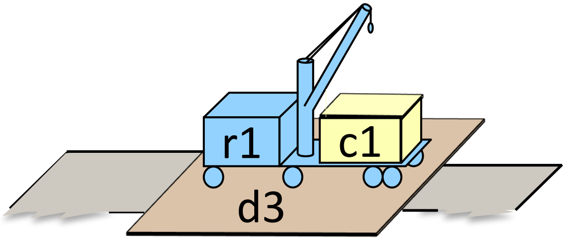


Example

- Computing $h^{\mathrm{FF}}(s_1)$
  - 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 $\hat{s}_k$ 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 $\hat{s}_0 = s_1$

Atoms in $\hat{s}_0 = s_1$:	Actions in A_1:	Atoms in $\hat{s}_1$:
loc(r1) = d1	move(r1,d1,d2)	loc(r1) = d2
loc(c1) = d1	move(r1,d1,d3)	**loc(r1) = d3**
cargo(r1) = nil	load(r1,c1,d1)	**loc(c1) = r1**
		cargo(r1) = c1
	from $\hat{s}_0$:	loc(c1) = d1
		loc(r1) = d1
		cargo(r1) = nil

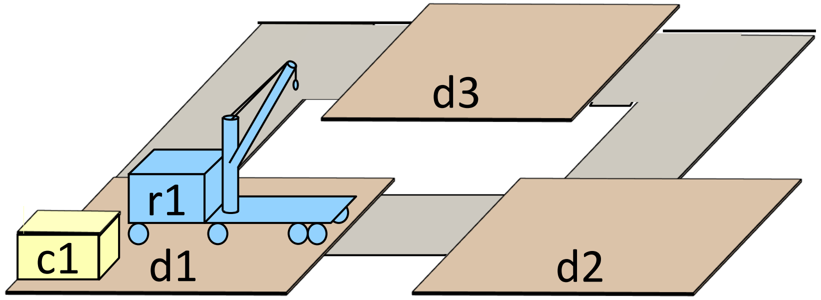
lines for preconditions and effects

$\hat{s}_1$ r-satisfies g, so $\langle A_1\rangle$ is a relaxed solution

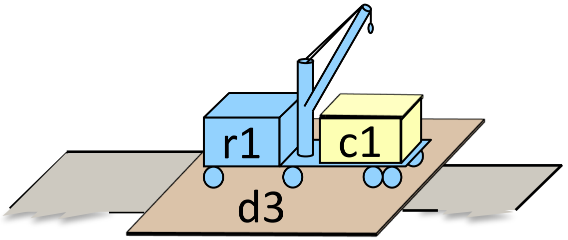
Example

- Computing $h^{\mathrm{FF}}(s_1)$

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

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

Atoms in $\hat{s}_0 = s_1$:	Actions in A_1:	Atoms in $\hat{s}_1$:
loc(r1) = d1	move(r1,d1,d2)	loc(r1) = d2
loc(c1) = d1	**move(r1,d1,d3)**	**loc(r1) = d3**
cargo(r1) = nil	**load(r1,c1,d1)**	**loc(c1) = r1**
		cargo(r1) = c1
$\hat{g}_0$	$\hat{a}_1$	$\hat{g}_1 = g$

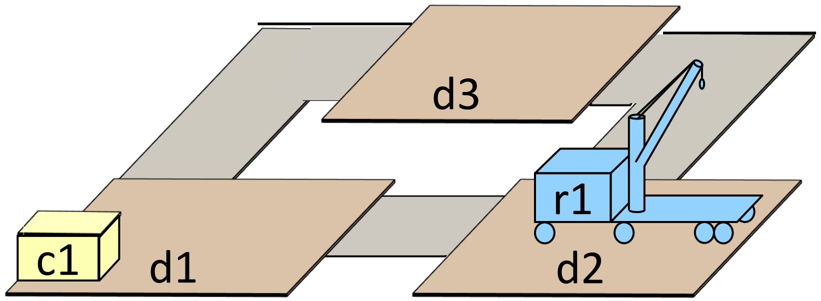
from $\hat{s}_0$:
loc(c1) = d1
loc(r1) = d1
cargo(r1) = nil

- $\hat{a}_1$ 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 $h^{\mathrm{FF}}(s_1) = 2$

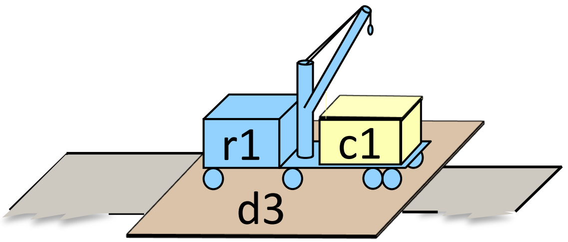
Example

- Computing $h^{\text{FF}}(s_2)$
  - 1. construct a relaxed solution
    - at each step, 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 $\hat{s}_k$ 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 ∞



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



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

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
- from $\hat{s}_0$:
  - 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)

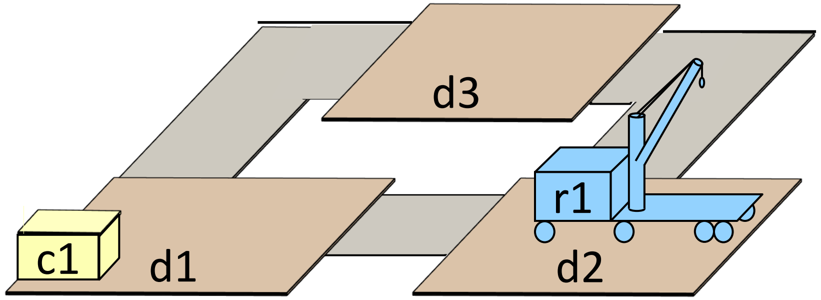
Atoms in $\hat{s}_2$:
- from $\hat{s}_1$:
  - loc(r1) = d2
  - loc(c1) = d1
  - cargo(r1) = nil
  - loc(r1) = d1
  - **loc(r1) = d3**
- cargo(r1) = c1
- **loc(c1) = r1**

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

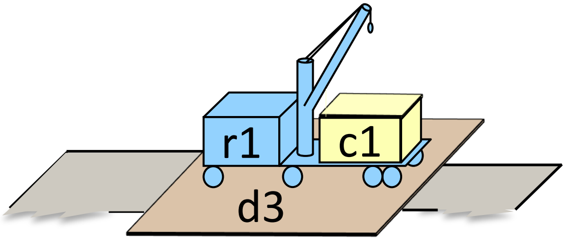
Example

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

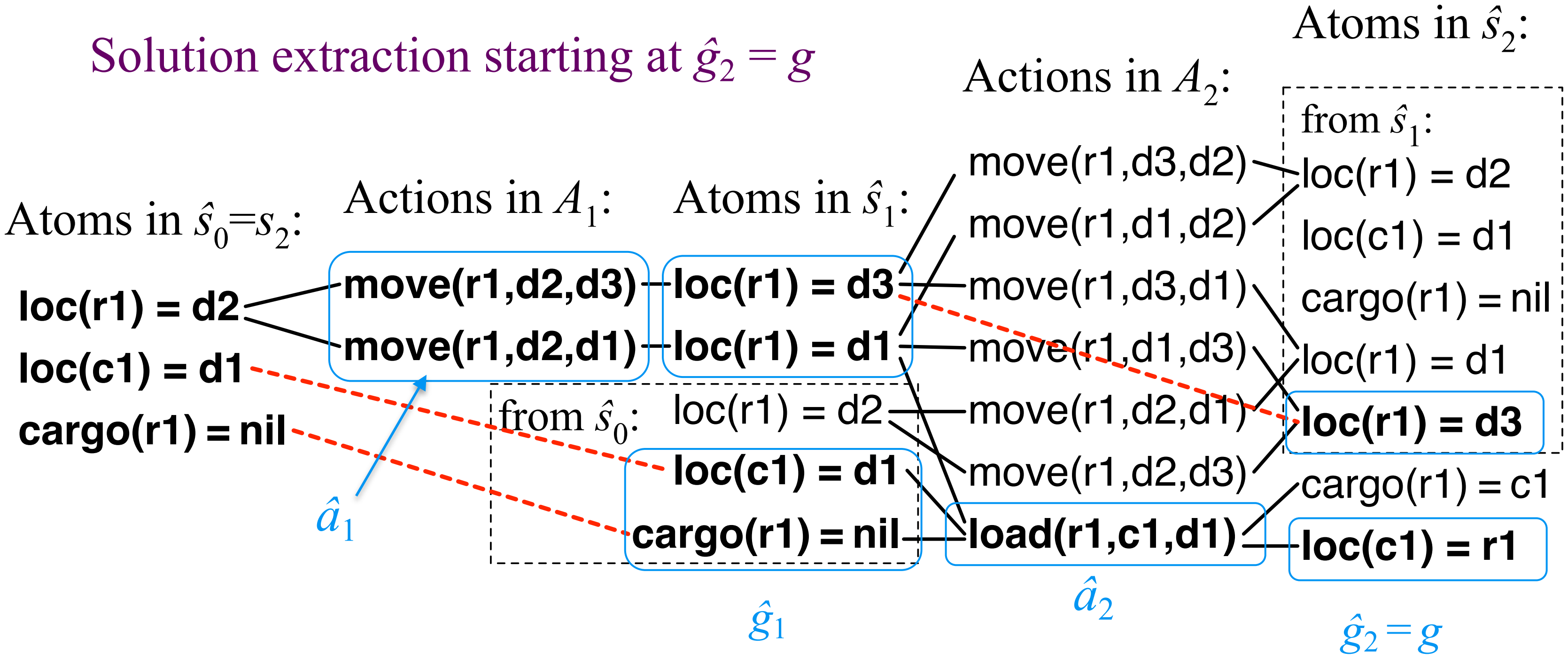


s_2 = {loc(r1)=d2, cargo(r1)=nil, loc(c1)=d2}



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)

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

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

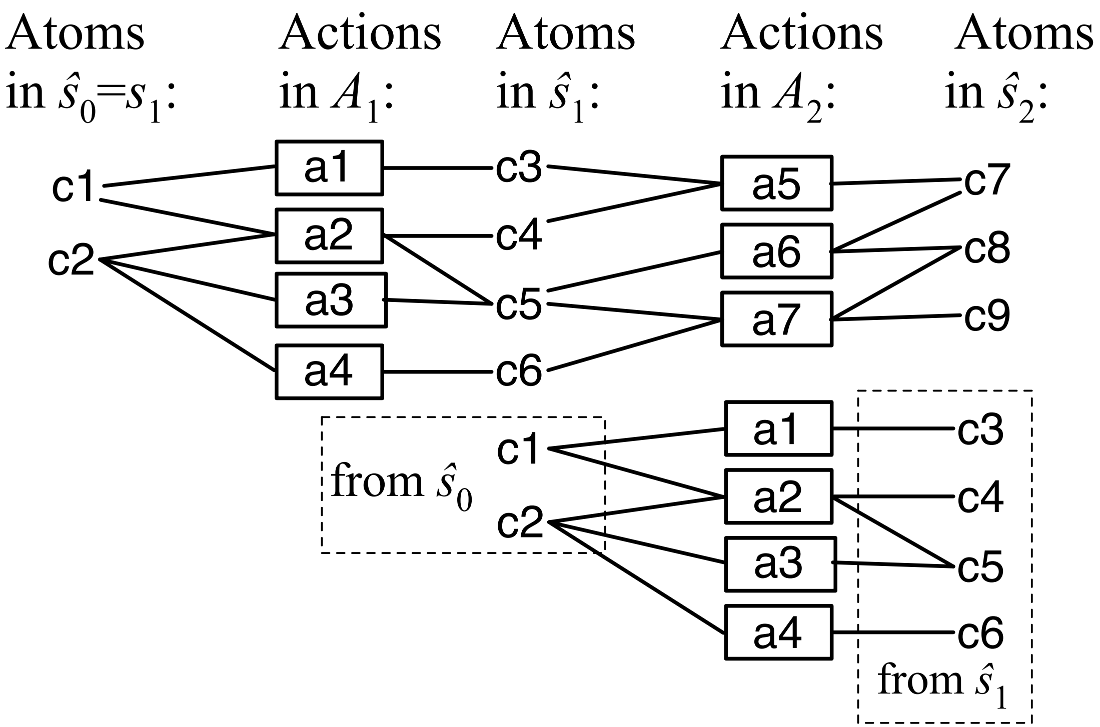
  - each $\hat{a}_i$ is a minimal set of actions such that $\gamma^+(\hat{s}_{i-1},\hat{a}_i)$ r-satisfies $\hat{g}_i$
    - *minimal* doesn't mean *smallest*

- $h^{\text{FF}}(s)$ 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

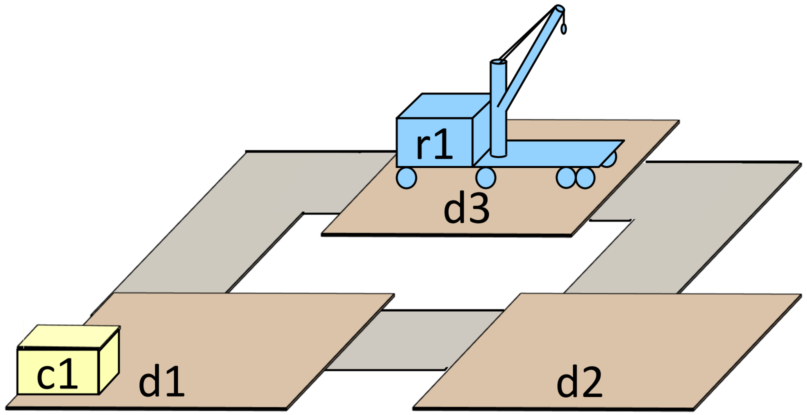


Poll. Suppose the goal atoms are c7, c8, c9. How many minimal relaxed solutions are there?

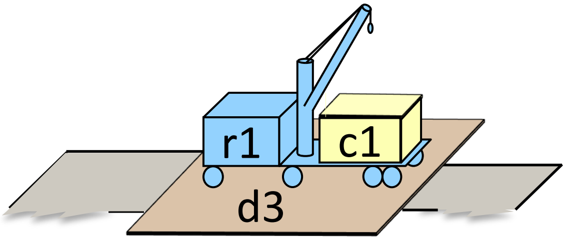
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. ≥ 8

2.3.3 Landmark Heuristics

- $P = (\Sigma, s_0, g)$ be a planning problem
- Let $\varphi = \varphi_1 \lor \ldots \lor \varphi_m$ 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}



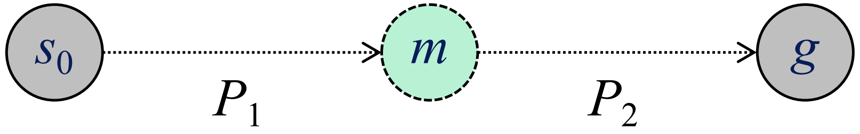
g = {loc(r1)=d3, loc(c1)=r1}

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

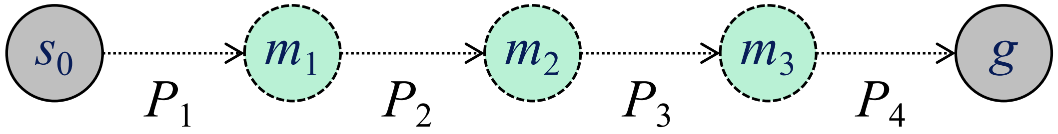
From now on, I'll abbreviate "disjunctive landmark" as "landmark"

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

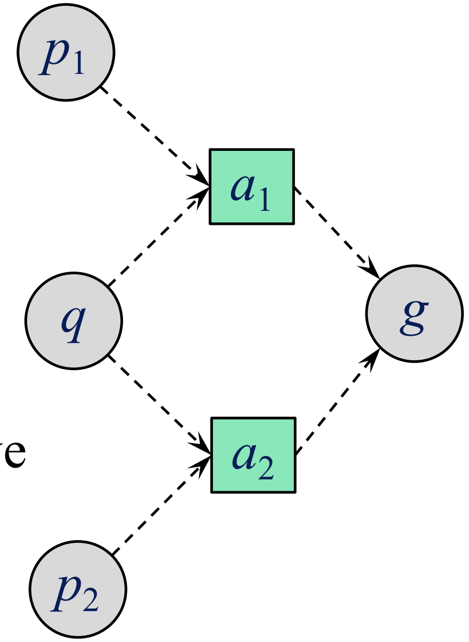


- Suppose m_1, m_2, m_3 are landmarks
  - Every solution to P must achieve m_1, then m_2, then m_3
- Possible strategy:
  - find a plan to go from s_0 to any state s_1 that satisfies m_1
  - find a plan to go from s_1 to any state s_2 that satisfies m_2
  - …

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
- Some landmarks are easier to find – polynomial time
  - Several procedures for finding them
  - I'll show you one based on relaxed planning graphs
- Why use RPGs?
  - Easier to solve relaxed planning problems
  - Easier to find landmarks for them
  - A landmark for a relaxed planning problem is also a landmark for the original planning problem

- Key idea: if φ is a landmark, get new landmarks from the preconditions of the actions that achieve φ
  - goal g
  - {actions that achieve g} $= \{a_1, a_2\}$
    - 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
    - $p_1 \vee p_2$



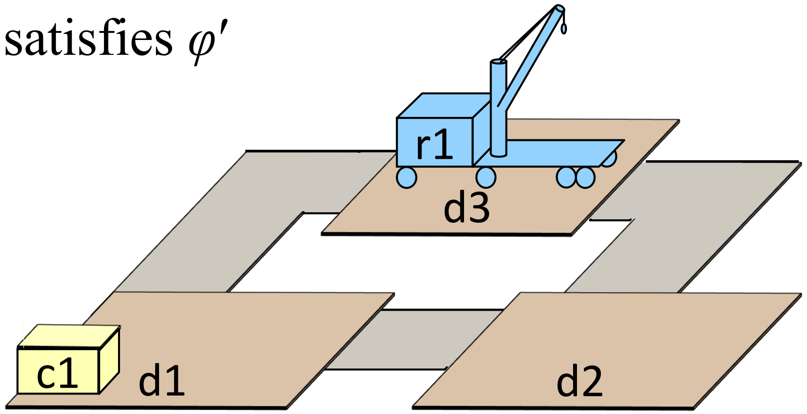
RPG-based Landmark Computation

- 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
- In this example, s_0 satisfies φ'

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

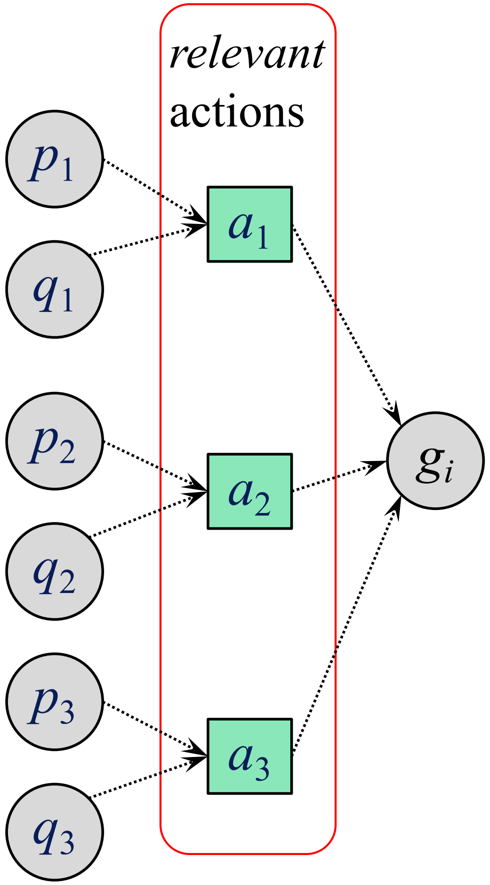
    $Preconds \leftarrow \bigcup\{\text{pre}(a) \mid a \in N\} \setminus s_0$

    $\Phi \leftarrow \{p_1 \vee p_2 \vee \ldots \vee p_m \mid m \leq 4$, 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

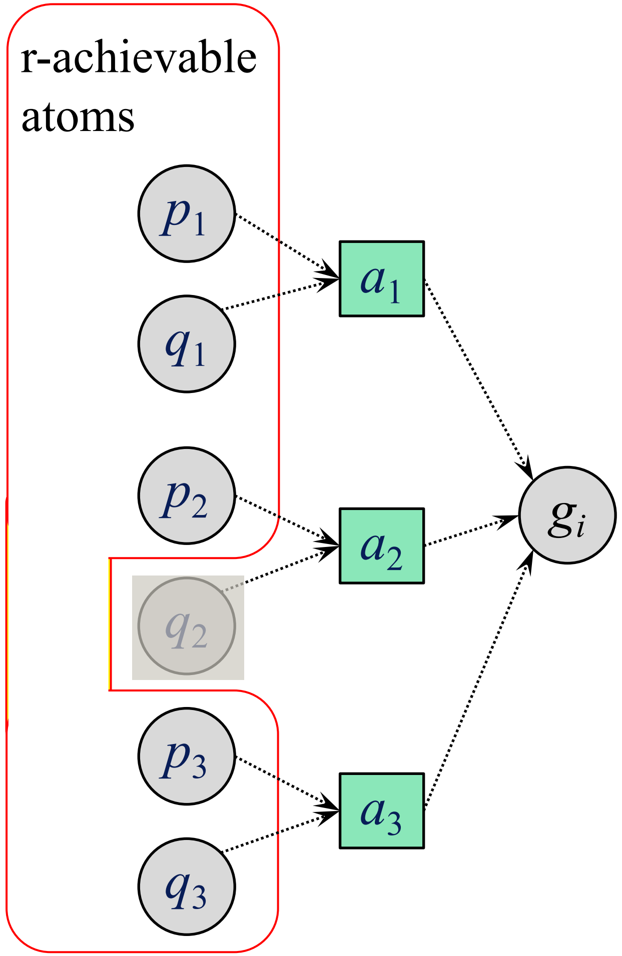
    Preconds $\leftarrow \bigcup\{\text{pre}(a) \mid a \in N\} \setminus s_0$

    $\Phi \leftarrow \{p_1 \vee p_2 \vee \ldots \vee p_m \mid m \leq 4$, 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, $g = \{g_1, g_2, \ldots, g_k\}$)

  $queue \leftarrow \{g_i \in g \mid s_0$ 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

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

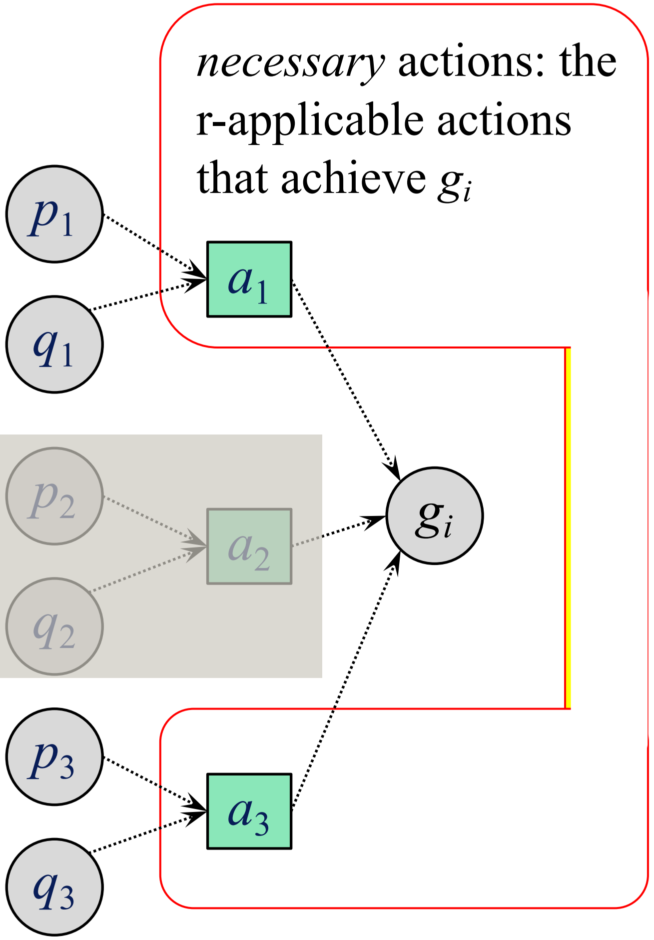
    $\Phi \leftarrow \{p_1 \vee p_2 \vee \ldots \vee p_m \mid m \leq 4$, every action in N has at least one p_i as a precondition, and every $p_i \in Preconds\}$

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

      add φ to $queue$

  return $Landmarks$

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



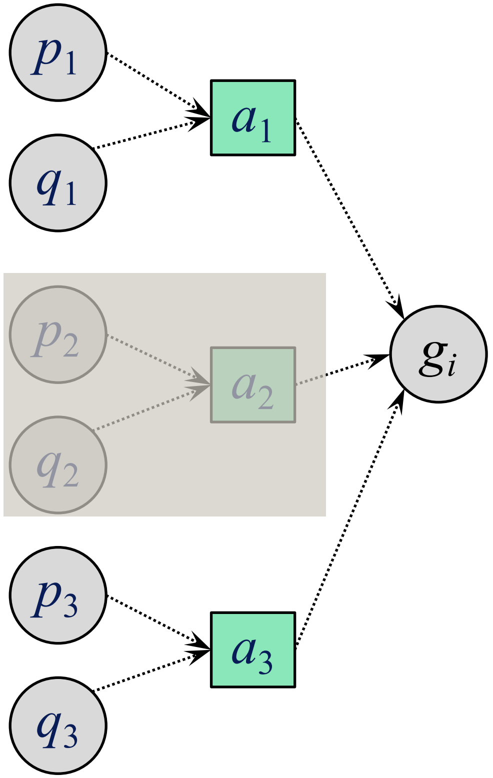
RPG-based Landmark Computation

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

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

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
  - $Preconds \leftarrow \bigcup\{\text{pre}(a) \mid a \in N\} \setminus s_0$
  - $\Phi \leftarrow \{p_1 \lor p_2 \lor \ldots \lor p_m \mid m \leq 4$, 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



Example

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

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

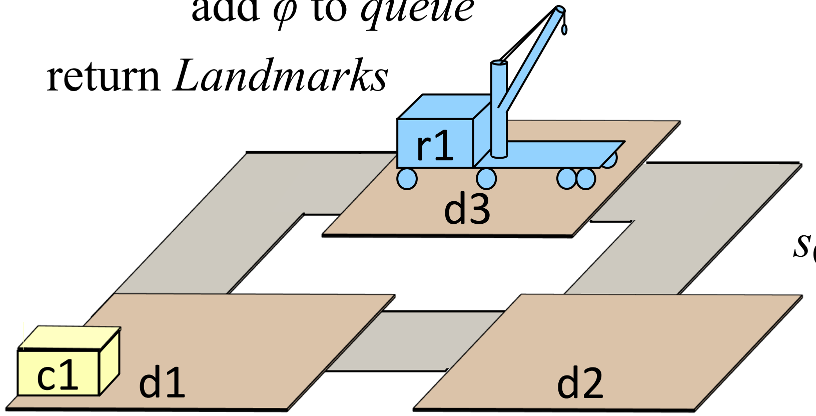
$\boxed{\begin{array}{l} queue = \langle \text{loc(c1)=r1} \rangle \\ Landmarks = \emptyset \end{array}}$

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

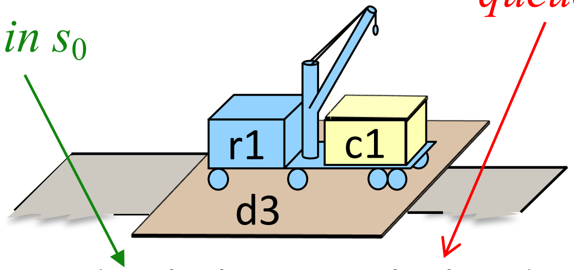
$r \in Robots$
$c \in Containers$
$l,d,e \in Locs$



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

true in s_0

add to *queue*



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

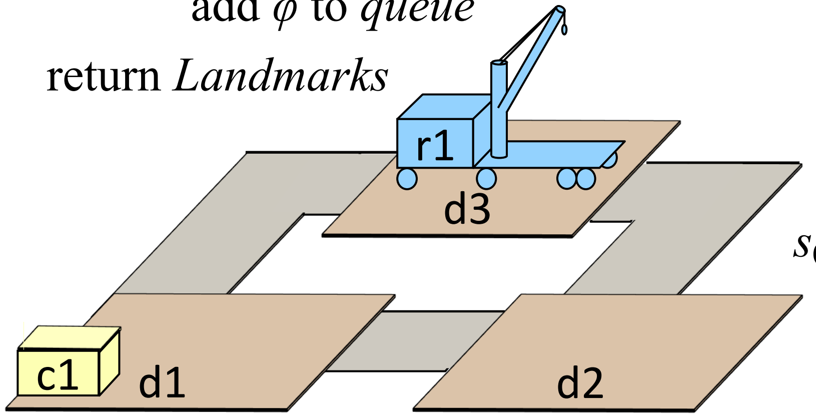
$Preconds \leftarrow \bigcup\{\text{pre}(a) \mid a \in N\} \setminus s_0$

$\Phi \leftarrow \{p_1 \vee p_2 \vee \ldots \vee p_m \mid m \leq 4$, 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$\}$

$queue = \langle\rangle$

g_i = loc(c1)=r1

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

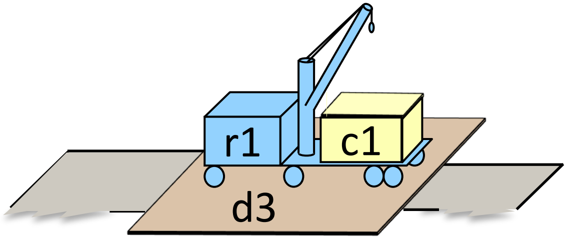
move(r, d, e)
- pre: loc(r)=d
- eff: loc(r)←e

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*
  - $R \leftarrow$ {actions whose effects include g_i}
  - if s_0 satisfies pre(a) for some $a \in R$ then return *Landmarks*
  - $\boxed{\text{generate RPG from } s_0 \text{ using } A \setminus R\text{, stopping when } \hat{s}_k = \hat{s}_{k-1}}$
  - $N \leftarrow$ {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*

queue = ⟨⟩

g_i = loc(c1)=r1

Landmarks = {loc(c1)=r1}

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

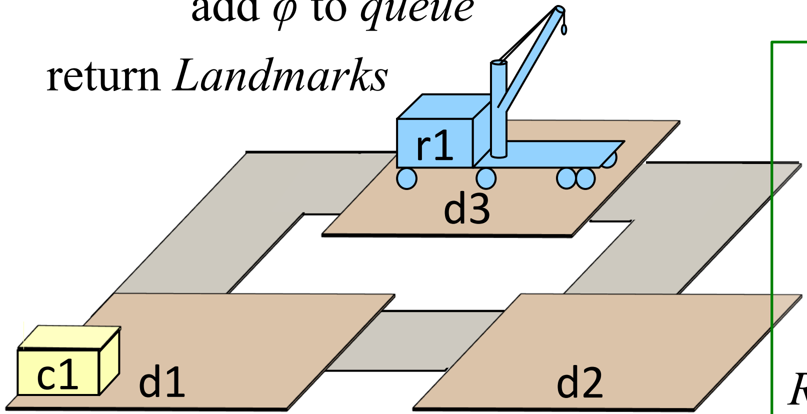
$A \setminus R$ = {the move and unload actions}

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

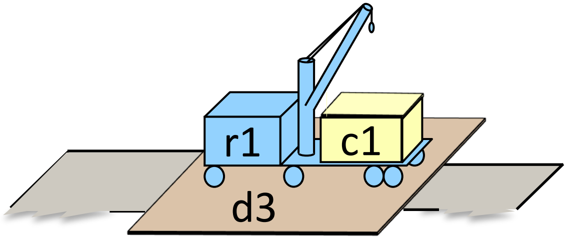
unload(r, c, l)
pre: loc(c)=r, loc(r)=l
eff: cargo(r)←nil, loc(c)←l

$r \in$ *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, $g = \{g_1, g_2, \ldots, g_k\}$)

- $queue \leftarrow \{g_i \in g \mid s_0$ 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 $Landmarks$

$queue = \langle\rangle$

g_i = loc(c1)=r1

$Landmarks$ = {loc(c1)=r1}

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

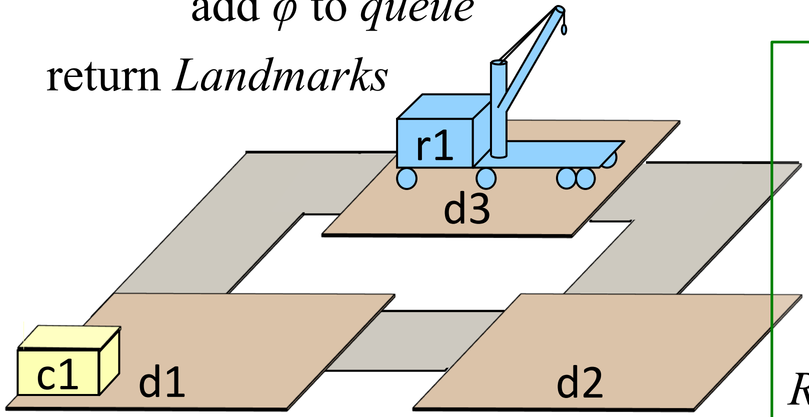
N = {load(r1,c1,d1)}

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

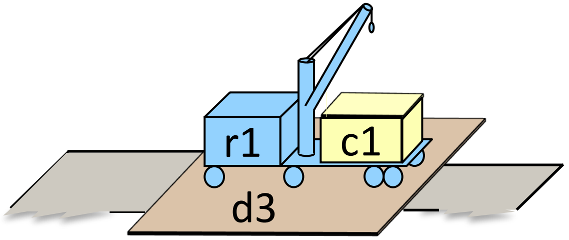
unload(r, c, l)
- pre: loc(c)=r, loc(r)=l
- eff: cargo(r)←nil, loc(c)←l

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



$\hat{s}_0$:	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 $A \setminus R$

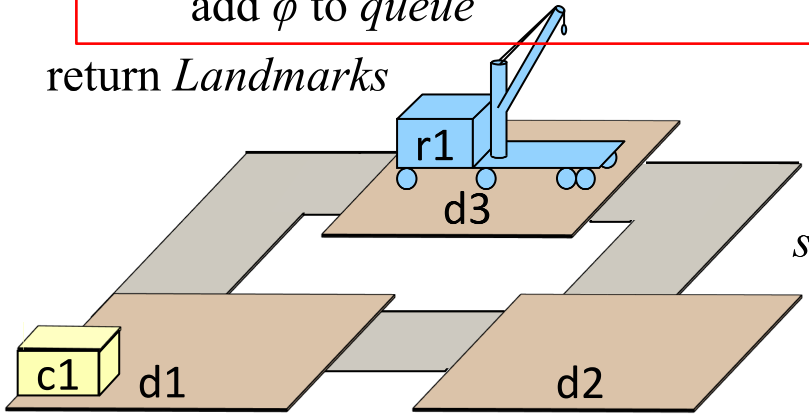


g = {loc(r1)=d3, loc(c1)=r1}

Example

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

  $queue \leftarrow \{g_i \in g \mid s_0$ 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}

$queue = \langle\rangle$

g_i = loc(c1)=r1

$Landmarks$ = {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

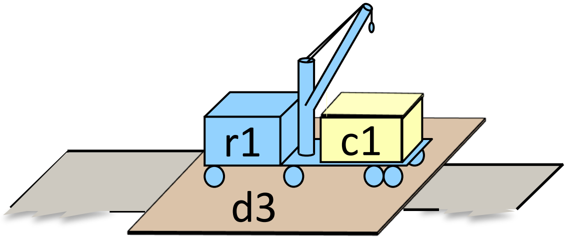
$Preconds$ = {loc(r1)=d1}
Φ = {loc(r1)=d1}
$queue$ = ⟨loc(r1)=d1⟩

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, $g = \{g_1, g_2, \ldots, g_k\}$)

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

  $\boxed{\text{while } queue \neq \emptyset}$

    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*

$queue = \langle \text{loc(r1)=d1} \rangle$

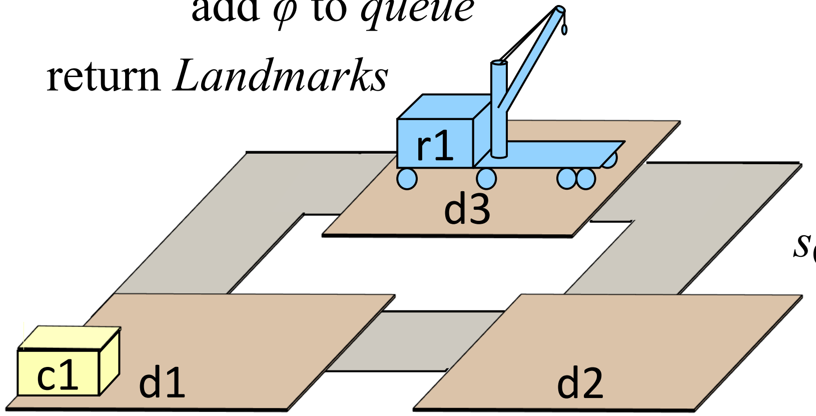
$Landmarks = \{\text{loc(c1)=r1}\}$

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

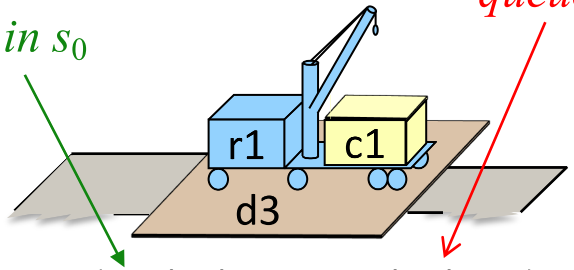
$r \in Robots$
$c \in Containers$
$l,d,e \in Locs$



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

add to *queue*

true in s_0



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

Example

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

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

$\quad$ while *queue* ≠ ∅

$\quad\quad$ remove a g_i from *queue*; add it to *Landmarks*

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

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

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

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

$\quad\quad$ if $N = \emptyset$ then return failure

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

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

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

$\quad\quad\quad$ add φ to *queue*

$\quad$ return *Landmarks*

queue = ⟨⟩

g_i = loc(c1)=r1

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

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

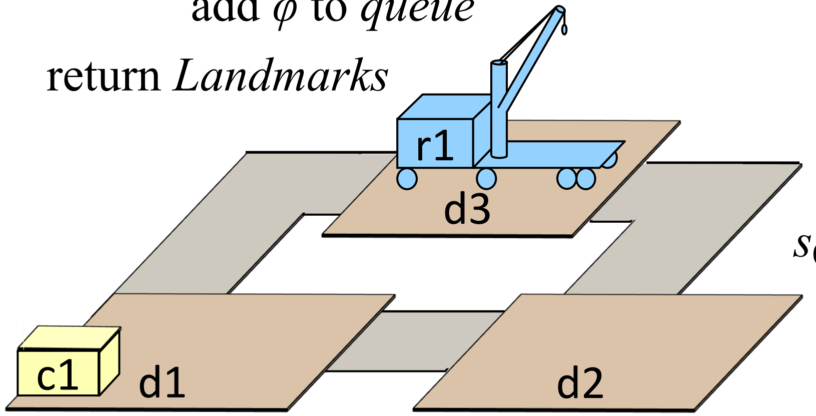
s_0 satisfies pre(move(r1,d3,d1))

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

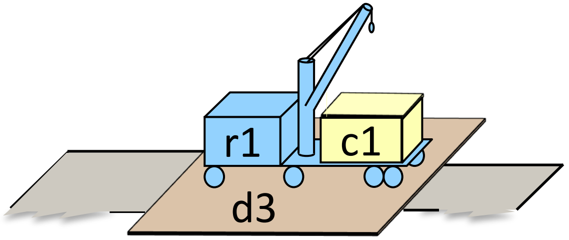
move(r, d, e)
- pre: loc(r)=d
- eff: loc(r)←e

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}

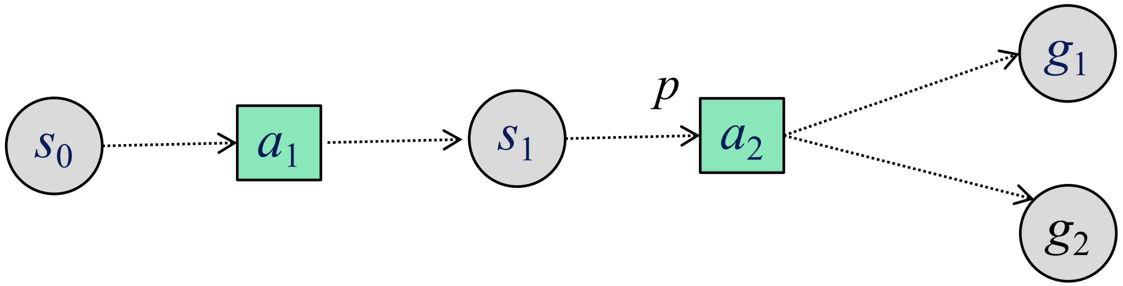
Landmark Heuristic

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

- **Poll**: Is this heuristic admissible?
  - 1. Yes 2. No

Landmark Heuristic

- Every solution to the problem needs to achieve all the computed landmarks
- One possible heuristic:
  - $h^{\mathrm{sl}}(s)$ = number of landmarks returned by RPG-Landmarks
- Not admissible



$g = \{g_1, g_2\}$

Three landmarks: g_1 , g_2 , p

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

Hierarchical Task Network (HTN) Planning

- For some planning problems, we may already have ideas for how to look for solutions
- Example: travel to a destination that's far away:
  - Brute-force search:
    - many combinations of vehicles and routes
  - Experienced human: small number of "recipes"

    e.g., flying:

    1. buy ticket from local airport to remote airport
    2. travel to local airport
    3. fly to remote airport
    4. travel to final destination

- Ways to put such information into a planner
  - Domain-specific algorithm
  - Domain-independent planning engine + domain-specific planning information
    - HTN planning (this section)
    - Control rules (Section 2.7.8)

- Similar idea for acting
  - Refinement methods (Chapter 3)

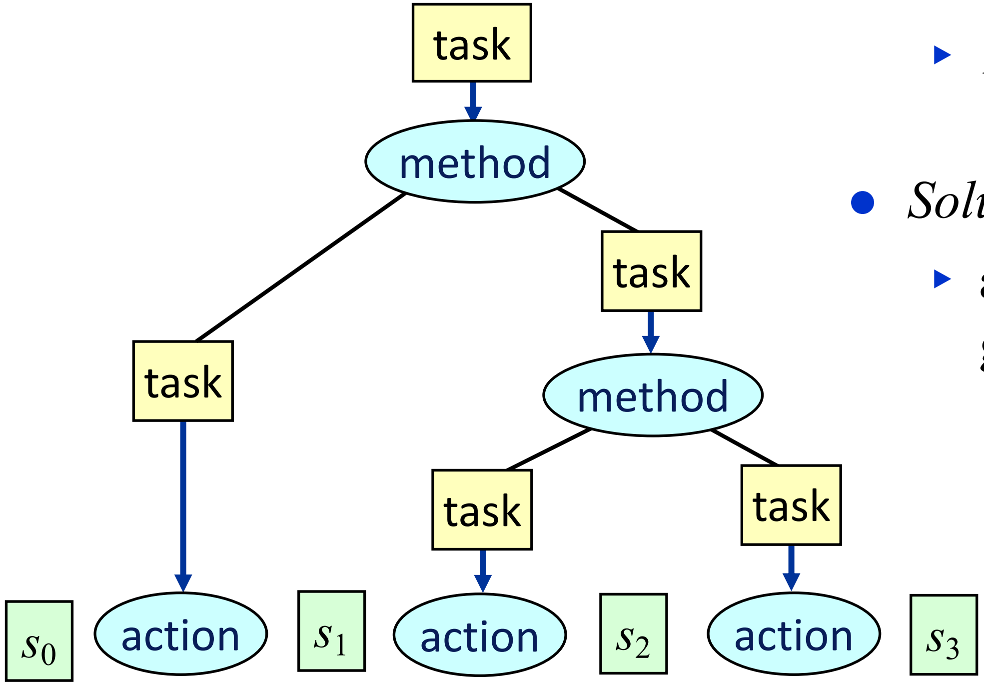
- Ingredients:
  - state-variable planning domain (Chap. 2)
  - *tasks*: activities to perform
  - *HTN methods*: ways to perform tasks

Total-Order HTN Planning

- Method format:

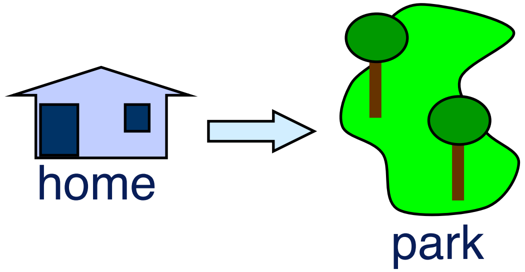
  method-name(*args*)
  Task: *task-name*(*args*)
  Pre: *preconditions*
  Sub: *list of subtasks and actions*

- *Primitive task*:
  - name of an action
- *Compound task*:
  - need to *decompose* (or *refine*) using methods



- HTN planning domain: a pair (Σ, M)
  - Σ: state-variable planning domain
  - M: set of methods
- Planning problem: $P = (\Sigma, M, s_0, T)$
  - T: list of tasks $\langle t_1, t_2, \ldots, t_k \rangle$
- *Solution*:
  - any executable plan that can be generated from T by applying
    - methods to nonprimitive tasks
    - actions to primitive tasks

Simple Travel-Planning Problem



- I'm at home, I have \$20, I want to go to a park 8 miles away

- s_0 = {loc(me)=home,
  cash(me)=20,
  dist(home,park)=8,
  loc(taxi)=elsewhere}

- Action templates:

walk (a,x,y)
 Pre: loc(a) = x
 Eff: loc(a) $\leftarrow y$

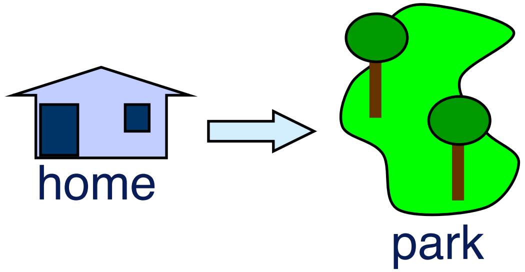
call-taxi (a,x)
 Pre: —
 Eff: loc(taxi) $\leftarrow x$,
  loc(a) $\leftarrow$ taxi

ride-taxi (a,x,y)
 Pre: loc(a) = taxi,
  loc(taxi) = x
 Eff: loc(taxi) $\leftarrow y$,
  owe(a) $\leftarrow 1.50 + ½$ dist(x,y)

pay-driver(a,y)
 Pre: owe(a) $\leq$ cash(a)
 Eff: cash(a) $\leftarrow$ cash(a) – owe(a),
  owe(a) $\leftarrow 0$,
  loc(a) = y

- Action parameters
  - $a \in$ *Agents*
  - $x,y \in$ *Locations*

Simple Travel-Planning Problem



- I'm at home, I have \$20, I want to go to a park 8 miles away
- *Task*: travel to the park
  - travel(me,home,park)

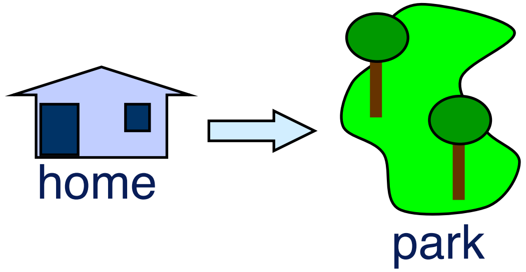
- *Methods*:

travel-by-foot(a,x,y)
- Task: travel(a,x,y)
- Pre: loc(a,x), distance(x, y) ≤ 4
- Sub: walk(a,x,y)

travel-by-taxi(a,x,y)
- Task: travel(a,x,y)
- Pre: loc(a,x), cash(a) $\geq 1.50 + ½$ dist(x,y)
- Sub: call-taxi (a,x), ride-taxi (a,x,y), pay-driver(a,y)

- Method parameters
  - $a \in Agents$
  - $x,y \in Locations$

Backtracking Search



travel(me,home,park)

travel-by-foot(me,home,park)
Pre:
✓ loc(me,home)
✗ dist(home,park) ≤ 4

Backtrack

travel-by-taxi(me,home,park)
Pre:
✓ loc(me,home)
✓ cash(me) ≥ 1.5+0.5*dist(home,park)

Solution plan:

⟨call-taxi(me,home),
ride-taxi(me,home,park),
pay-driver(me)⟩

s_0
Initial state

loc(me) = home
cash(me) = 20
dist(home,park) = 8
loc(taxi) = station

call-taxi(me,home)

Pre: …
Eff: …

loc(me) = taxi
cash(me) = 20
dist(home,park) = 8
loc(taxi) = home

ride-taxi(me,home,park)

Pre: …
Eff: …

s_2

loc(me) = taxi
cash(me) = 20
dist(home,park) = 8
loc(taxi) = park
owe(me) = 5.50

pay-driver(me,park)

Pre: …
Eff: …

s_3
Final state

loc(me) = park
cash(me) = 14.5
dist(home,park) = 8
loc(taxi) = park
owe(me) = 0

Total-Order HTN Planning Algorithm

- TFD(s, T, π)
  - if $T = \langle\rangle$ then return π
  - let $t_1, t_2, \ldots, t_k$ be the tasks in T — i.e., $T = \langle t_1, t_2, \ldots, t_k \rangle$
  - if t_1 is primitive then
    - if there is an action a such that head(a) matches t_1 and a is applicable in s:
      - return TFD($\gamma(s,a)$, $\langle t_2,\ldots,t_k \rangle$, $\pi . a$)
    - else: return failure
  - else // t_1 is nonprimitive
    - for each $m \in M$:
      - if task(m) matches t_1 and m is applicable in s:
        - $\pi \leftarrow$ seek-plan(s, subtasks(m) . $\langle t_2,\ldots,t_k \rangle$, π)
        - if $\pi \neq$ failure then return π
    - return failure

- The SHOP algorithm
  - http://www.cs.umd.edu/projects/shop/
  - Depth-first, left-to-right search

- For each primitive task, apply action

state s; $T = \langle a, t_2, \ldots, t_k \rangle$

new state $\gamma(s,a)$; new $T = \langle t_2, \ldots, t_k \rangle$

- For each compound task, decompose

state s, $T = \langle t_1, t_2, \ldots, t_k \rangle$

method instance m

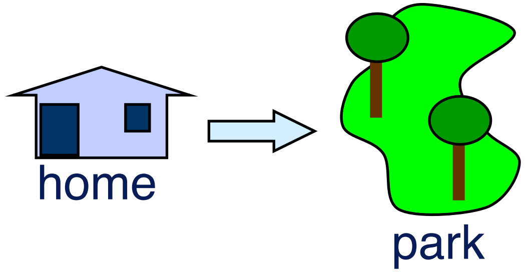
new $T = \langle \overbrace{u_1, \ldots, u_j}^{m}, t_2, \ldots, t_k \rangle$

Pyhop

- A simple HTN planner written in Python
  - Implements a version of TFD
  - Works in both Python 2.7 and 3.2
- State: Python object that contains variable bindings
  - To say taxi is at park in state s, write
    - s.loc['taxi'] = 'park’
- Actions and methods: ordinary Python functions
- Some limitations compared to most other HTN planners
  - I’ll discuss later
- Open-source software, Apache license
  - http://bitbucket.org/dananau/pyhop
- Simple travel example
  - download Pyhop

```
import simple_travel_example
```

States



- State:
  - Python object that holds state-variable bindings
- State variables:
  - loc[*x*], cash[*x*], owe[*x*], dist[*x*][*y*]

```
state1 = pyhop.State('initial state')
state1.loc = {'me':'home'}
state1.cash = {'me':20}
state1.owe = {'me':0}
state1.dist = {'home':{'park':8}, 'park':{'home':8}}
```

- Python dictionary notation for state1.loc['me'] = 'home', etc.

Operators (i.e., Actions)

- Written as Python functions

```
def walk(s,a,x,y):          # s is the current state
    if s.loc[a] == x:       # Preconditions are if-tests
        s.loc[a] = y        # Modify the state
        return s            # Return the modified state
    else: return False      # Action is inapplicable

def call_taxi(state,a,x):
    state.loc['taxi'] = x
    return state
```

- Similar definitions for ride_taxi, pay_driver

- Tell Pyhop what the actions are:

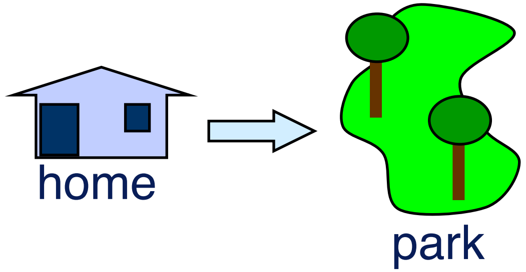
```
pyhop.declare_operators(walk, call_taxi, ride_taxi, pay_driver)
```

Tasks and Methods

- Task: n-tuple ($taskname$, arg_1, …, arg_n)
  - e.g., ('travel', 'me', 'home', 'park')
- Method: Python function

```
def travel_by_foot(s,a,x,y):          # s is the current state
   if s.dist[x][y] <= 4:              # precondition
      return [('walk',a,x,y)]         # return subtask list
   return False                       # inapplicable => return False

def travel_by_taxi(s,a,x,y):
   if s.cash[a] >= 1.5 + 0.5*s.dist[x][y]:   # precondition
      return [('call_taxi',a,x),              # return subtask list
            ('ride_taxi',a,x,y),
            ('pay_driver',a,x,y)]
   return False                               # inapplicable

pyhop.declare_methods('travel', travel_by_foot, travel_by_taxi)
```

Planning Problems, Solutions



- Planning problem:

```
pyhop(state1, [('travel','me','home','park')])
```

- Solution plan:

```
[('call_taxi','me','home’),
 ('ride_taxi','me','home','park’),
 ('pay_driver','me')]
```

Comparison

- Task: transport a container *c*
  - Pseudocode for an HTN method

```
Method m_transport(r,x,c,y,z)
    Task: transport(c,y,z)
    Pre: loc(r) = x, cargo(r) = nil, loc(c) = y
    Sub: move(r,x,y), take(r,c,y), move(r,y,z), put(r,c,z)
```

- Most HTN planners (e.g., SHOP):
- Write in a planning language the planner can read and analyze
- Can have parameters not mentioned in the task (e..g, *r* and *x* above)
  - Can backtrack over multiple method instances
- Planner knows in advance what the subtasks will be
  - Helps with implementing heuristic functions

- Pyhop method: ordinary Python function

```
def m_transport(c,y,z):
    if loc(r) == x and cargo(r) == nil and loc(c) == y:
        (r,x) = find_suitable_robot('transport',c,y,z)
        return [move(r,x,y), take(r,c,y), move(r,y,z), put(r,c,z)]
    else: return False
```

- Advantages
  - Don't need to learn a planning language
  - Can do complex reasoning to evaluate preconditions, generate subtasks
- Disadvantages:
  - Don't know in advance what the subtasks are
    - How to implement a heuristic function?
  - How to implement uninstantiated parameters?

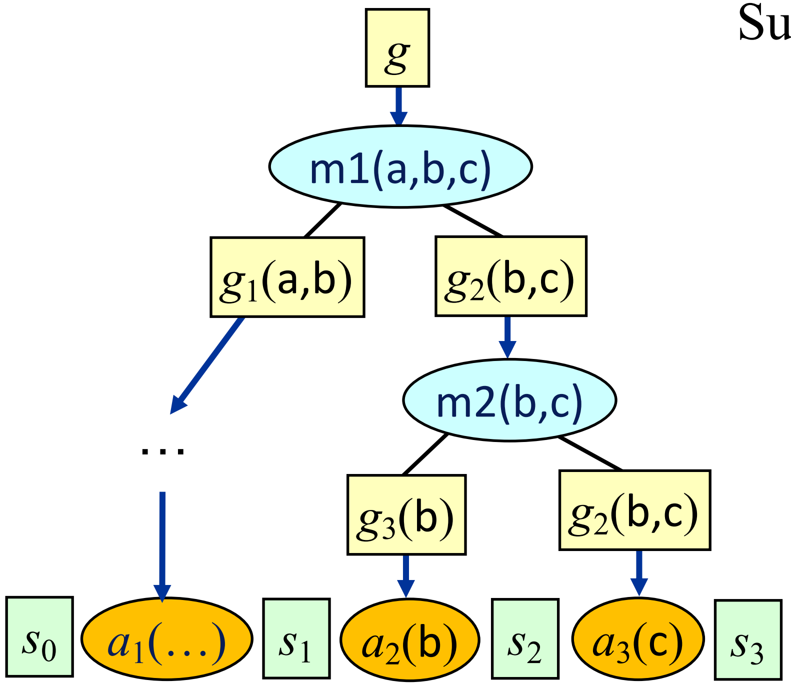
Total-Order Hierarchical Goal Network (HGN) Planning

- Like HTN planning, but with goals instead of tasks
- HGN planning domain: a pair (Σ, M)
  - Σ: state-variable planning domain
    - Same states, actions as in HTN planning
  - M: set of HGN methods
- Format for HGN methods:

  method-name(*parameters*)
  ~~Goal: *goal formula*~~ ← unneeded
  Pre: *preconditions*
  Sub: *list of subgoals*

m1(x_1, x_2, x_3)
Pre: $p_1(x_1), p_2(x_2)$
Sub: $g_1(x_1,x_2), g_2(x_2,x_3)$

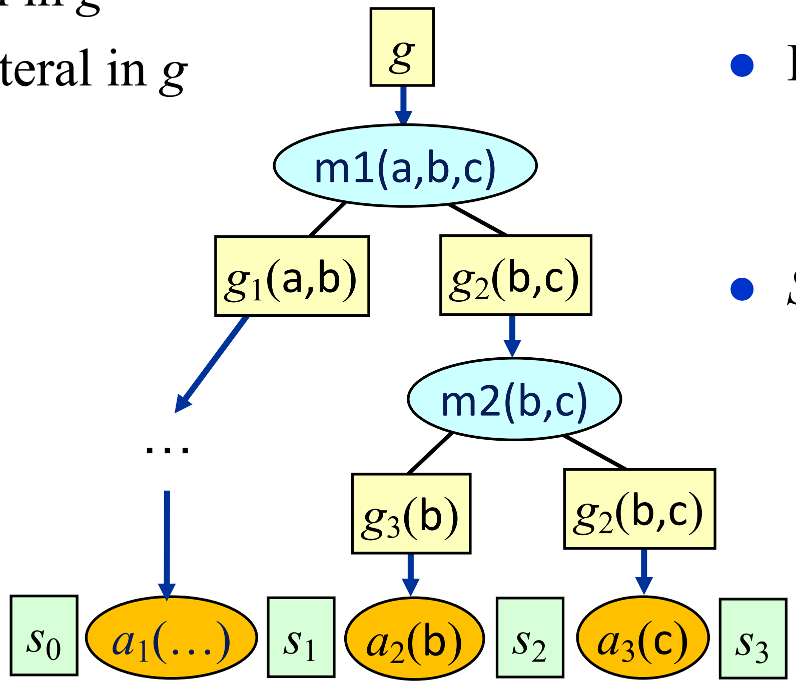
m2(y_1, y_2)
Pre: $q_1(y_1, y_2)$
Sub: $g_3(y_1), g_2(y_1,y_2)$



- m's preconditions: $\text{pre}(m) = \{p_1, \ldots, p_j\}$
- m's subgoals: $\text{sub}(m) = \langle g_1, \ldots, g_k \rangle$
  - each g_i is a set of literals

Total-Order Hierarchical Goal Network (HGN) Planning

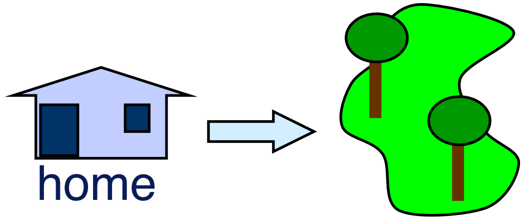
- Let m be a method instance
- m's *postcondition*, post(m), is m's last subgoal
  - A set of literals that m will make true
- m is *relevant* for a goal g if
  - post (m) $\vDash$ at least one literal in g
  - post(m) $\nvDash$ negation of any literal in g
- an action a is *relevant* for g if
  - post (m) $\vDash$ at least one literal in g
  - post(m) $\nvDash$ negation of any literal in g



- m is applicable in a state s if $s \vDash$ pre(m)
  - Same as for an action
- HGN planning problem:

  $P = (\Sigma, M, s_0, G)$
  - (Σ, M) is an HGN planning domain
  - $G = \langle g_1, g_2, \ldots, g_n \rangle$ is a list of goals
- Each g_i is a set of ground literals
  - Like a goal in a classical planning problem
- *Solution* for P:
  - any plan that we can get by applying methods and actions that are both relevant and applicable

Simple Travel-Planning Problem



- I'm at home, I have \$20, I want to go to a park 8 miles away

- s_0 = {loc(me)=home,
  cash(me)=20,
  dist(home,park)=8,
  loc(taxi)=elsewhere}

- Action templates:

walk (a,x,y)
  Pre: loc(a) = x,
    distance(x, y) ≤ 4
  Eff: loc(a) $\leftarrow y$

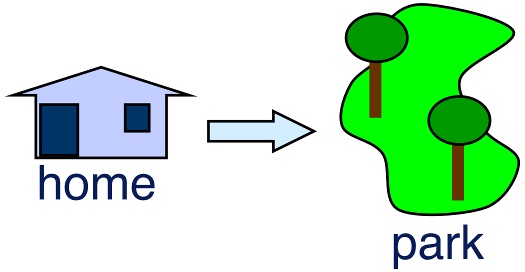
call-taxi (a,x)
  Pre: —
  Eff: loc(taxi) $\leftarrow x$,
    loc(a) $\leftarrow$ taxi

ride-taxi (a,x,y)
  Pre: loc(a) = taxi,
    loc(taxi) = x
  Eff: loc(taxi) $\leftarrow y$,
    owe(a) $\leftarrow 1.50 + \frac{1}{2}$ dist(x,y)

pay-driver(a,y)
  Pre: owe(a) $\leq$ cash(a)
  Eff: cash(a) $\leftarrow$ cash(a) – owe(a),
    owe(a) $\leftarrow 0$,
    loc(a) = y

- Action parameters
  - $a \in$ *Agents*
  - $x,y \in$ *Locations*

Simple Travel-Planning Problem



- I'm at home, I have \$20, I want to go to a park 8 miles away

- *Goal*: be in the park
  - loc(me) = park

- *HGN Methods*:

~~travel-by-foot(a,x,y)~~

~~Pre: loc(a,x),~~
~~distance(x, y) $\leq$ 4~~

~~Sub: walk(a,x,y)~~

travel-by-taxi(a,x,y)

Pre: loc(a,x),
cash(a) $\geq 1.50 + \frac{1}{2}$ dist(x,y)

Sub: loc(a) = taxi,
loc(taxi) = y,
loc(a) = y

- Method parameters
  - $a \in$ *Agents*
  - $x,y \in$ *Locations*

Total-Order HGN Planning Algorithm

- GDP(s, G, π)
  - if $G = \langle\rangle$ then return π
  - $g \leftarrow$ the first goal formula in G
  - if $s \vDash g$ then
    - remove g from G; return GDP(s, G, π)
    - $U \leftarrow$ {actions and method instances that are relevant for g and applicable in s}
    - if $U = \emptyset$ then return failure
    - nondeterministically choose $u \in U$
    - if u is an action then
      - append u to π; $s \leftarrow \gamma(s,u)$
    - else insert sub(u) at the front of G
    - return GDP(s, G, π)

- The GDP algorithm
  - Depth-first, left-to-right search

state s; $G = \langle g_1, g_2, \ldots, g_k\rangle$

u is an action

new state $\gamma(s,u)$; G doesn't change

state s; $G = \langle g_1, g_2, \ldots, g_k\rangle$

u is a method instance

new $G = \langle \overbrace{g_{u1}, \ldots, g_{uj}}^{u}, g_1, g_2, \ldots, g_k\rangle$

GTPyhop

- GTPyhop (2021):
- Like Pyhop, but plans for both tasks and goals
  - declare *task methods* for accomplishing tasks
  - declare *goal methods* for achieving goals
- Open-source: https://github.com/dananau/GTPyhop

- HTN planning mostly backward-compatible with Pyhop
- Example in the GTPyhop software distribution:
  - Examples/pyhop_simple_travel_example
- Near-verbatim version of the Pyhop simple travel example
  - Documentation tells what the differences are

- HGN planning is similar to GDP, but not identical

- In GDP, *relevance* for a goal depends on either eff(a) or the last element of sub(m)
  - Uses this to decide whether to execute a or m
- Problem: to get eff(a) or sub(m), GTPyhop must execute a or m
  - No way to know sub(m) until then

- Work-around:
  - For each method m, declare what goals it's relevant for
  - If m is relevant for g, require it to accomplish every literal in g (instead of just some of them)
  - Don't allow actions to be relevant for goals, but allow them to appear in sub(m)

Planning Algorithm

- Augment the Pyhop algorithm to plan for both tasks and goals
- *To-do* list: actions, tasks, goals
- For each goal
  - use a goal method to decompose it into a todo list
  - add a dummy action that will fail if the goal isn't achieved (guarantees soundness)
- Whenever there's a failure
  - Backtrack to nearest task or goal, look for a different method
  - If there isn't one, backtrack further

- Input: state s; to-do list T
- if T is empty, return π
- case(first element of T):
  - **action**: *apply it*, append it to π, call planner recursively on the rest of T
  - **task**: find a relevant task method, *apply it*, call planner recursively on new T
  - **goal**: find a relevant goal method, *apply it*, call planner recursively on new T

state s; $T = [a, t_2, \ldots, t_k]$

new state $\gamma(s,a)$; new $T = [t_2, \ldots, t_k]$

state s, $T = [\,\tau, t_2, \ldots, t_k]$

task method m

new $T = [\overbrace{u_1, \ldots, u_j}^{m}, t_2, \ldots, t_k]$

state s, $T = [\,g, t_2, \ldots, t_k]$

goal method m

new $T = [\overbrace{u_1, \ldots, u_j, \mathit{verify}(g)}^{m}, t_2, \ldots, t_k]$

- Check whether g is true
- If it isn't, then fail (GTPyhop will backtrack)

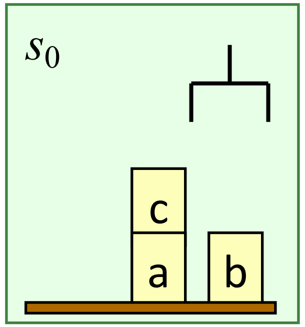
Example: Blocks World

- Simple classical planning domain
  - Blocks, robot hand for stacking them, infinitely large table

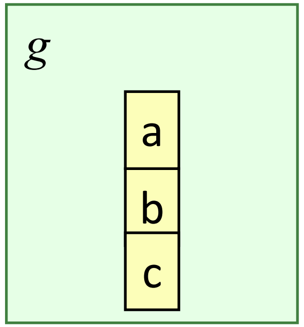
- pickup(x)
  - pre: loc(x)=table, clear(x)=T, holding=nil
  - eff: loc(x)=crane, clear(x)=F, holding=x
- putdown(x)
  - pre: holding=x
  - eff: holding=nil, loc(x)=table, clear(x)=T
- unstack(x,y)
  - pre: loc(x)=y, clear(x)=T, holding=nil
  - eff: loc(x)=crane, clear(x)=F, holding=x, clear(y)=T
- stack(x,y)
  - pre: holding=x, clear(y)=T
  - eff: holding=nil, clear(y)=F, loc(x)=y, clear(x)=F

- The "Sussman anomaly"
  - Planning problem that caused problems for early classical planners

s_0 = {clear(a)=F, clear(b)=T,
clear(c)=T,
loc(a)=table,
loc(b)=table, loc(c)=a,
holding(hand)=nil}



g = {loc(a)=b, loc(b)=c}

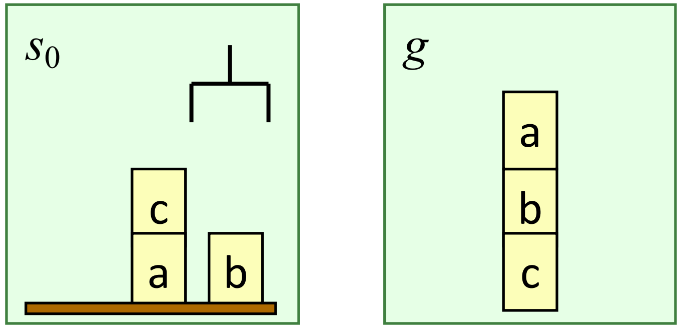


π = ⟨unstack(c,a), putdown(c),
pickup(b), stack(b,c),
pickup(a), stack(a,b)⟩

Domain-Specific Algorithm

loop

  if there's clear block that needs to be moved
    and it can immediately be moved to a place
    where it won't need to be moved again
    then move it there
  else if there's a clear block that needs to be moved
    then move it to the table
  else if the current state satisfies the goal
    then return success
  else return failure



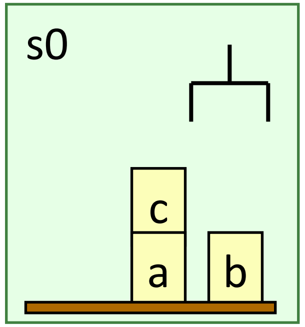
π = ⟨unstack(c,b),
putdown(c),
pickup(b),
stack(b,c),
pickup(a),
stack(a,b)⟩

- Situations in which c needs to be moved:
  - loc(c)=d, goal contains loc(c)=e, and $d \neq e$
  - loc(c)=d, d is a block, goal contains loc(b)=d, and $b \neq c$
  - loc(c)=d and d is a block that needs to be moved
- Can extend this to include situations involving clear and holding

- Sound, complete, guaranteed to terminate
- Runs in time $O(n^3)$
  - Can be modified to run in time $O(n)$
- Often finds optimal (shortest) solutions, but sometimes only near-optimal
  - For block-stacking problems, PLAN-LENGTH is NP-complete
- Can implement as GTPyhop methods

States and goals

- Initial state:

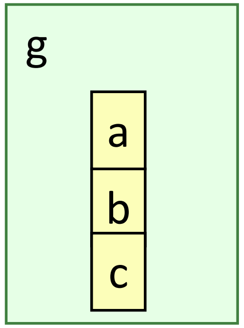


- A State object to hold all the state-variable bindings:

```
s0 = gtpyhop.State('Sussman initial state')
s0.pos = {'a':'table', 'b':'table', 'c':'a'}
s0.clear = {'a':False, 'b':True, 'c':True}
s0.holding = {'hand':False}
```

  - Python dictionary notation for `s0.pos['a'] = 'table'`, etc.

- Goal:




```
g = gtpyhop.Multigoal('Sussman goal')
g.pos = {'a':'b', 'b':'c'}
```

- Two kinds of goals:
  - *Unigoal*: a triple (*name, arg, value*)
    - represents a desired state-variable binding
    - e.g., unigoal ('pos', 'a', 'b’)
      - find a state s in which s.pos['a'] == 'b'
  - *Multigoal*: state-like object
    - represents a conjunction of unigoals
    - g: find a state s in which
      - s.pos['a'] == 'b' and s.pos[‘b'] == 'c'

Actions

- Blocks-world pickup action
  - **if** *x* is on table, *x* is clear, and robot hand is empty
  - **then** modify *s* and return it
  - **else** return nothing
    - means the action isn't applicable
    - also OK to return false like Pyhop does

- putdown action – similar

- Easy to write similar "stack" and "unstack" actions

• Args: current state *s*, block *x*

```
def pickup(s,x):
    if s.pos[x] == 'table' \
      and s.clear[x] == True \
      and s.holding['hand'] == False:
    s.pos[x] = 'hand'
    s.clear[x] = False
    s.holding['hand'] = x
return s

def putdown(s,x):
    if s.holding['hand'] = x:
        s.pos[x] = 'table'
        s.clear[x] = True
        s.holding['hand'] = False
    return s
```

Preconditions: *if* test

Effects: modify variable bindings in *s*

• Tell GTPyhop these are actions

```
gtpyhop.declare_actions(pickup,putdown)
```

Task methods

- m_take: method to pick up a clear block *x*, regardless of what it's on
  - Args: current state *s*, block *x*.
  - if *x* is clear:
    - Return one to-do list if *x* is on the table, another to-do list if *x* isn't on the table
  - Else return nothing
    - means method is inapplicable
    - (also OK to return false like Pyhop does)

- Last line says m_take is a task method
  - relevant for all tasks of the form (take, ...)

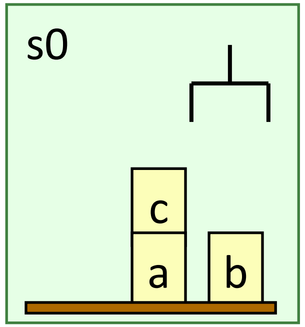
- m_put: similar, for all tasks of the form (put, ...)

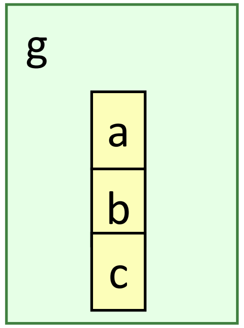
```
def m_take(s,x):
    if s.clear[x] == True:
        if s.pos[x] == 'table':
            return [('pickup', x)]
        else:
            return [('unstack',x,s.pos[x])]

gtpyhop.declare_task_methods('take',m_take)

def m_put(s,x,y):
    if s.holding['hand'] == x:
        if y == 'table':
            return [('putdown',x)]
        else:
            return [('stack',x,y)]
    else:
        return False

gtpyhop.declare_task_methods('put',m_put)
```

Goal methods

loop

  if there’s clear block that needs to be moved
    and it can immediately be moved to a place
    where it won’t need to be moved again
    then move it there
  else if there’s a clear block that needs to be moved
    then move it to the table
  else if the current state satisfies the goal
    then return success
  else return failure






```
def m_moveblocks(s, mgoal):
    for x in all_clear_blocks(s):
        stat = status(x, s, mgoal)
        if stat == 'move-to-block':
            where = mgoal.pos[x]
            return [('take',x), ('put',x,where), mgoal]
        elif stat == 'move-to-table':
            return [('take',x), (put,x,'table'), mgoal]
        for x in all_clear_blocks(s):
            if status(x,s,mgoal) == 'waiting' \
                    and s.pos[x] != 'table':
                return [('take',x), ('put',x,'table'), mgoal]
        return [ ]
```

- s = current state
- mgoal = a multigoal
- red = helper functions

```
gtpyhop.declare_multigoal_methods(m_moveblocks)
```

```
find_plan(s0,g)
```

returns `[('unstack','c','a'), ('putdown','c'), ('pickup','b'), ('stack','b','c'), ('pickup','a'), ('stack','a','b')]`

Discussion

- Earlier we discussed limitations/strengths of Pyhop compared to most other HTN planners
  - Same discussion also applies to GTPyhop
- Similar comparison for GTPyhop vs. most HGN planners

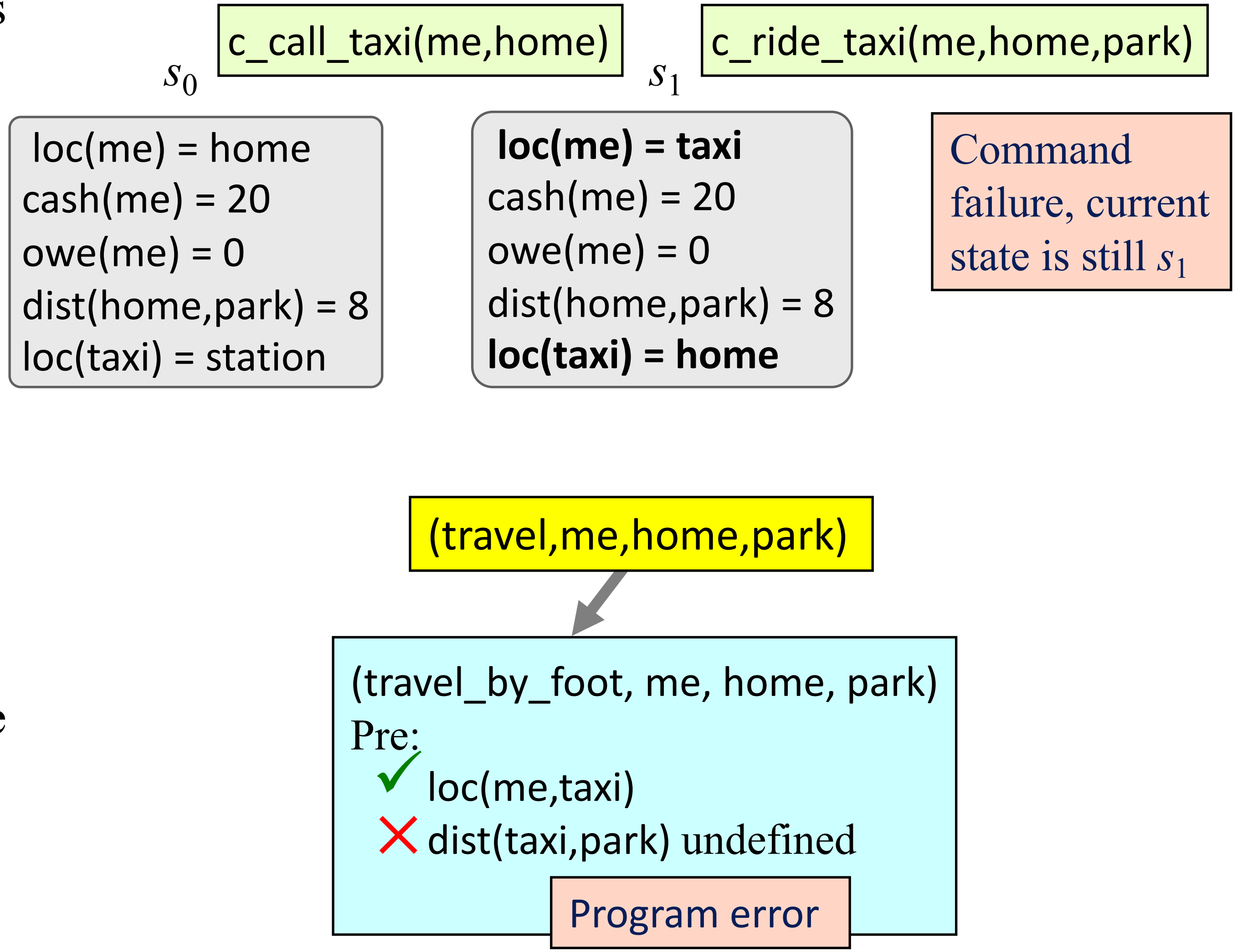
Acting and Planning

- run_lazy_lookahead(*state, todo_list*)
  - loop:
    - *plan* = find_plan(*state, todo_list*)
    - if *plan* = [ ]:
      - return *state*   // the new current state
    - for each *action* in *plan*:
      - execute the corresponding command
      - if the command fails:
        - continue the outer loop

- Simple Travel Problem:
  - run_lazy_lookahead calls
    - find_plan(s_0, [(travel,me,home,park)])
  - find_plan returns
    - [(call_taxi,me,home),
      (ride_taxi,me,home,park),
      (pay_driver,me)]
  - run_lazy_lookahead executes
    - c_call_taxi(me,home)
    - c_ride_taxi(me,home,park)
    - c_pay_driver(me)
- If everything executes correctly, I get to the park
  - But suppose the taxi breaks down …

Acting and Planning

- For planning and acting, need to HTN methods that can recover from unexpected problems
- Example:
  - run_lazy_lookahead executes
    - c_call_taxi(me,home)
    - c_ride_taxi(me,home,park)
      - Suppose the 2nd command fails
  - run_lazy_lookahead calls
    - find_plan(s_1, [(travel,me,home,park)])
      - Error: tries to use an undefined value
- To run this example in GTPyhop:
  - import Examples.simple_htn_acting_error



Summary

- Total-order HTN planning
  - Planning problem: initial state, list of *tasks*
  - Apply HTN *methods* to tasks to get *subtasks* (smaller tasks)
    - Do this recursively to get smaller and smaller subtasks
      - At the bottom: *primitive tasks* that correspond to actions
  - Search goes down and forward
- Pyhop: Python implementation of total-order HTN planning
  - Open source: http://bitbucket.org/dananau/pyhop
- GTPyhop: Python implementation of HTN + HGN planning
  - Open source: https://github.com/dananau/GTPyhop
- Examples: simple travel, blocks world
- To integrate planning and acting, need to make sure the HTN methods can handle unexpected events
  - One way: make GTPyhop re-entrant

Search Direction, Search Strategies

- Down, then forward
  - totally-ordered tasks: find-plan, SHOP, Pyhop
  - partially-ordered tasks: SHOP2, SHOP3
  - goals instead of tasks: GDP, GoDeL
  - acting, task refinement: RAE (Chap. 3)
  - Monte Carlo rollouts: UPOM
- Down and backward
  - plan-space planning: SIPE, O-Plan, UMCP
- Forward, then down (level 1, level 2, level 3, …)
  - AHA*: A* search
  - Bridge Baron 1997: game-tree generation