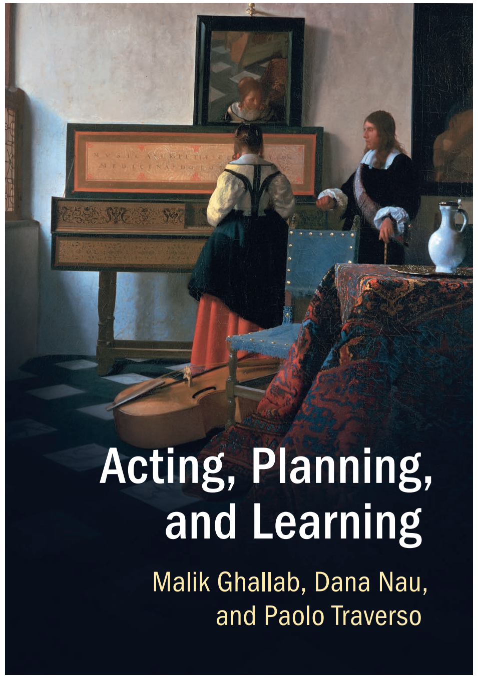
Last update: 11:09 PM, March 7, 2025



Chapter 3
Planning with Deterministic Models

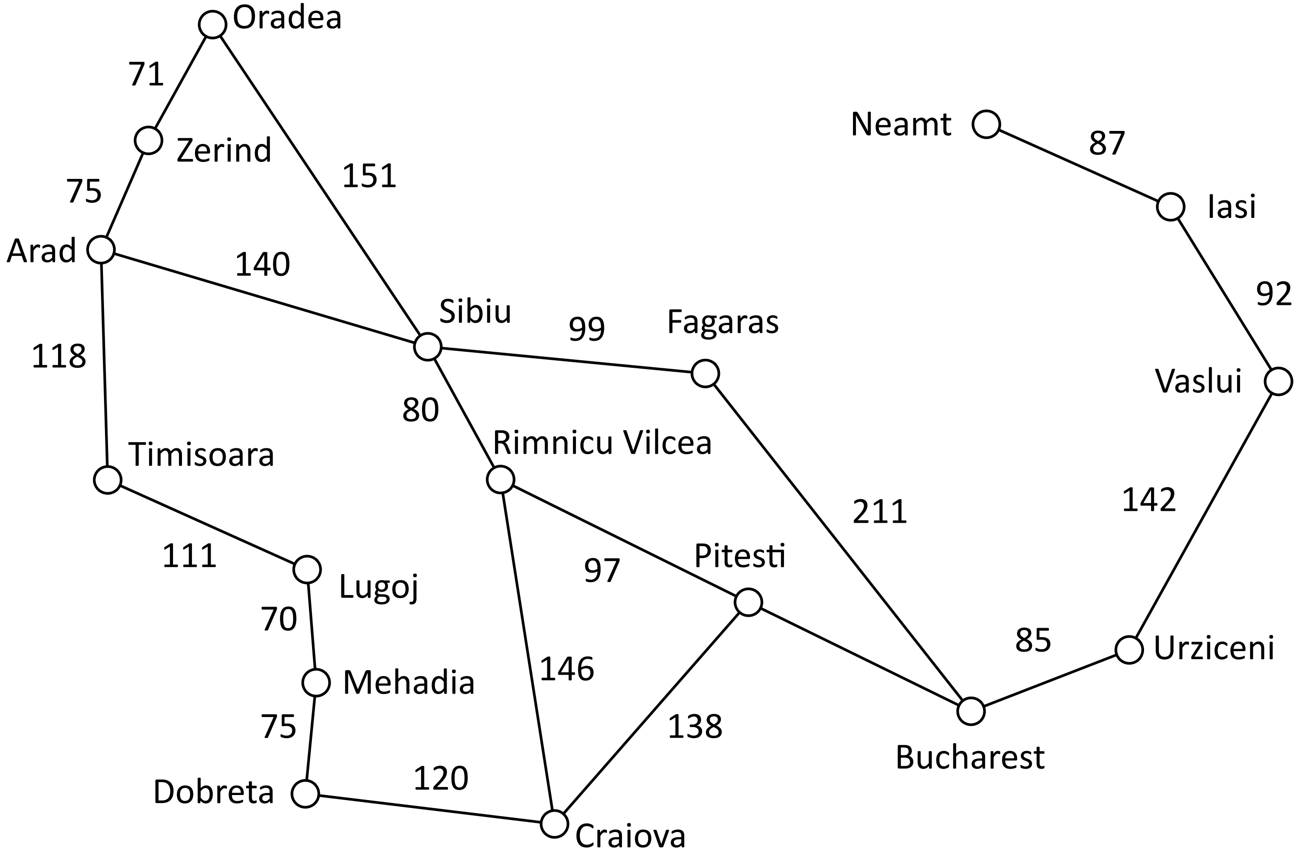
3.1. Forward State-Space Search
3.2. Heuristic Functions

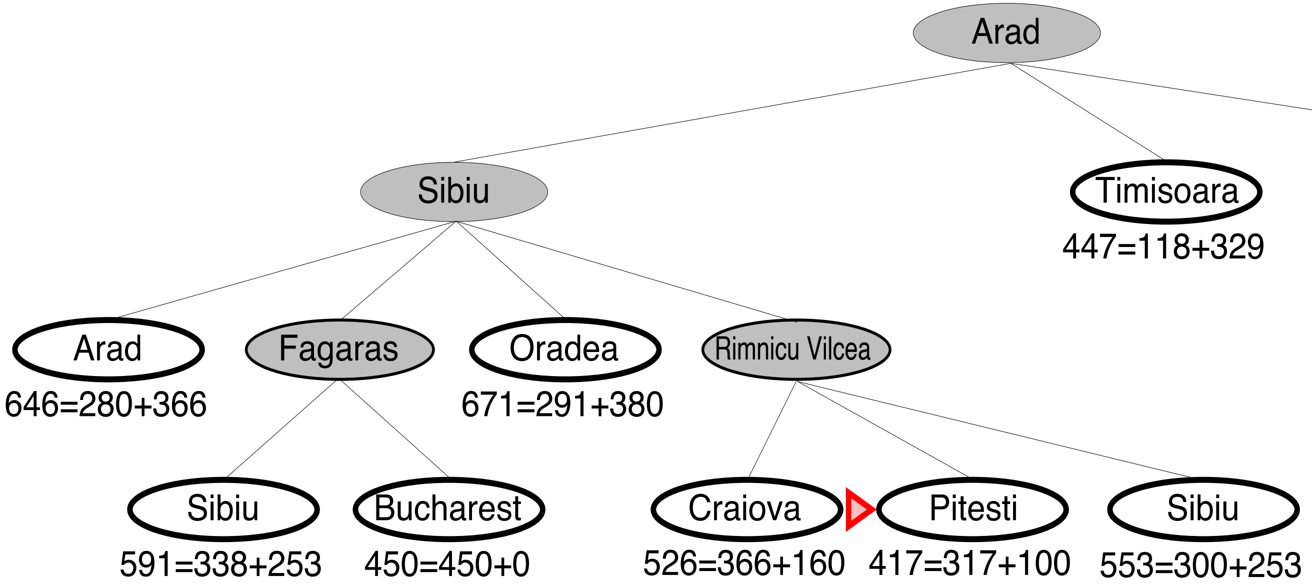
Dana S. Nau
University of Maryland

with contributions from
Mark "mak" Roberts

Planning as Search

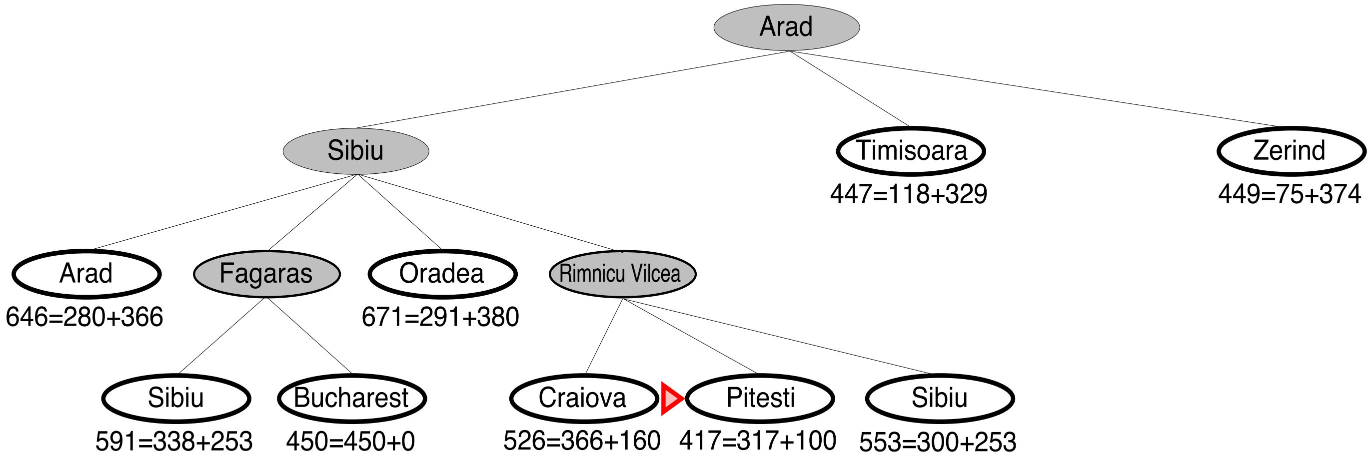
- Most AI planning procedures are search procedures
  - *Search tree*: the data structure the procedure uses to keep track of which paths it has explored





Credit: Stuart Russell, lecture slides for *Artificial Intelligence: A Modern Approach*

Search-Tree Terminology



- *Node* ≈ a pair $\nu = (\pi, s)$, where $s = \gamma(s_0, \pi)$
  - In practice, ν will contain other things too
    - depth(ν), cost(π), pointers to parent and children, …
  - π isn't always stored explicitly, can be computed from the parent pointers
- *children* of $\nu = \{(\pi.a, \gamma(s,a)) \mid a \text{ is applicable in } s\}$
- *successors* or *descendants* of ν:
  - children, children of children, etc.
  - sometimes called a subtree
- *ancestors* of ν
  = {nodes that have *v* as a successor}
- *initial* or *starting* or *root* node $\nu_0 = (\langle\rangle, s_0)$
  - root of the search tree
- *path* from the root node: sequence of nodes $\langle \nu_0, \ldots, \nu_n \rangle$ such that each ν_i is a child of ν_{i-1}
- *height* of search space
  = length of longest acyclic path from ν_0
- *depth* of ν
  = length(π) = length of path from ν_0 to ν
- *branching factor* of ν
  = number of children of ν
- *branching factor* of a search tree
  = max branching factor of the nodes
- *expand* ν: generate all children

3.1. Forward Search

Forward-search (Σ, s_0, g)

$s \leftarrow s_0$; $\pi \leftarrow \langle\rangle$

while $s \not\models g$ **do**

if $Applicable(s) = \emptyset$ **then return** failure   ← *Dead end*

nondeterministically choose $a \in Applicable(s)$

$s \leftarrow \gamma(s,a)$; $\pi \leftarrow \pi \cdot a$

return π

- Nondeterministic algorithm
  - *Sound*: if an execution trace returns a plan π, it's a solution
  - *Complete*: if the planning problem is solvable, at least one of the possible execution traces will return a solution
- Represents a class of deterministic search algorithms
  - Deterministic versions of the nondeterministic choice
    - Which leaf node to expand next
    - Which nodes to prune from the search space
  - They'll all be sound, but not necessarily complete

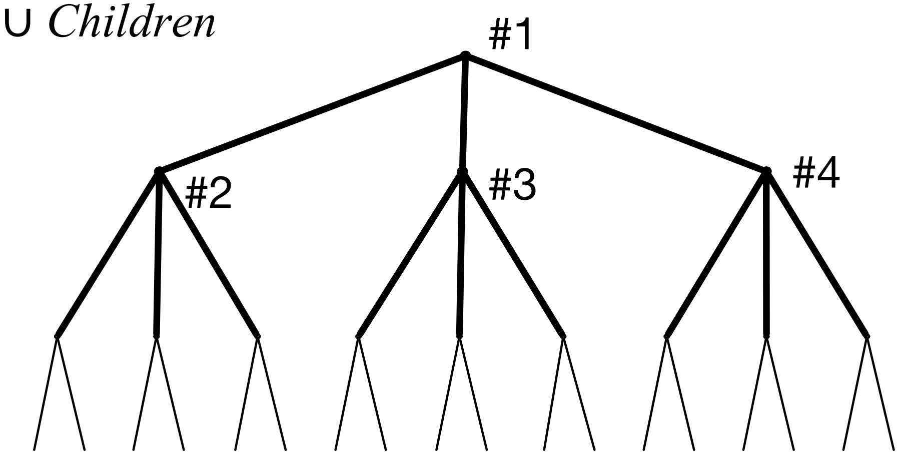
Deterministic Version

Forward-Search-Det(Σ, s_0, g)
    Frontier ← {(⟨⟩, s_0)}
    Expanded ← ∅
    while *Frontier* ≠ ∅ **do**
(*i*)   select a node ν = (π, s) ∈ *Frontier*
        remove ν from *Frontier*
        add ν to *Expanded*
        if *s* satisfies *g* **then return** *π*
        Children ← {($\pi \cdot a$, $\gamma(s,a)$) | $a \in$ *Applicable*(s)}
(*ii*)   prune 0 or more nodes from
            Children, *Frontier*, *Expanded*
        Frontier ← *Frontier* ∪ *Children*
    return failure

- Special cases:
  - depth-first, breath-first, A*, many others
- Classify by
  - how they *select* nodes (*i*)
  - how they *prune* nodes (*ii*)
- Pruning often includes *cycle-checking*:
  - Remove from *Children* every node (π,s) that has an ancestor (π',s') such that $s' = s$
- In classical planning problems, S is finite
  - Cycle-checking will guarantee termination

Breadth-First Search (BFS)

Forward-Search-Det(Σ, s_0, g)
  Frontier ← {(⟨⟩, s_0)}
  Expanded ← ∅
  while *Frontier* ≠ ∅ **do**
(i) select a node ν = (π, s) ∈ *Frontier*
  remove ν from *Frontier*
  add ν to *Expanded*
  if s satisfies g **then return** π
  Children ← {($\pi \cdot a$, $\gamma(s,a)$) | a ∈ *Applicable*(s)}
(ii) prune 0 or more nodes from
   Children, *Frontier*, *Expanded*
  Frontier ← *Frontier* ∪ *Children*
  return failure



(i): Select (π,s) ∈ *Frontier* that has the smallest length(π), i.e., smallest number of edges

- Possible tie-breaking rules:
  - left-to-right
  - select smallest $h(s)$ - will discuss later

(ii): Remove every (π,s) ∈ *Children* ∪ *Frontier* such that s ∈ *Expanded*

- Thus expand states at most once

- Properties
  - Terminates
  - Returns solution if one exists
    - shortest, but not least-cost
  - Worst-case complexity:
    - memory $O(|S|)$, running time $O(b|S|)$
      - b = max branching factor
      - $|S|$ = number of states in S

Depth-First Search (DFS)

Forward-Search-Det(Σ, s_0, g)

    $Frontier \leftarrow \{(\langle\rangle, s_0)\}$

    $Expanded \leftarrow \emptyset$

    while $Frontier \neq \emptyset$ **do**

(i)   select a node $\nu = (\pi, s) \in Frontier$

        remove ν from *Frontier*

        add ν to *Expanded*

        if s satisfies g **then return** π

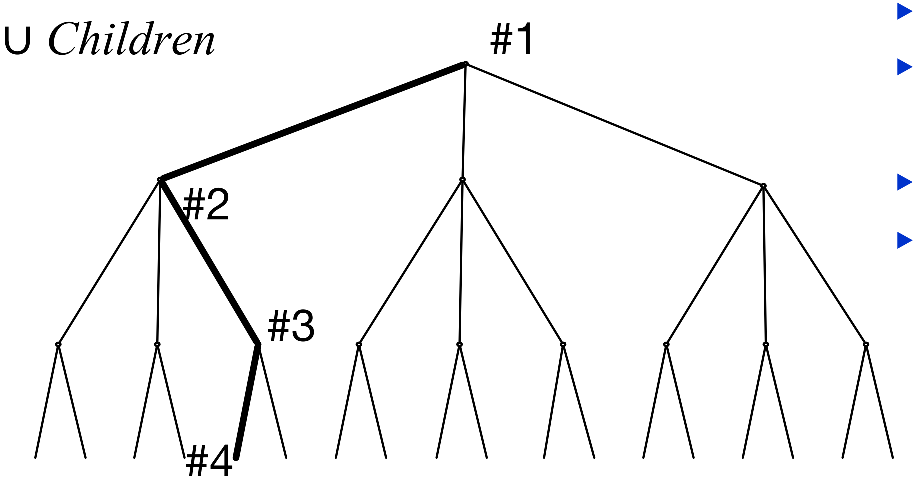
        $Children \leftarrow \{(\pi \cdot a, \gamma(s,a)) \mid a \in Applicable(s)\}$

(ii)   prune 0 or more nodes from

            Children, *Frontier*, *Expanded*

        $Frontier \leftarrow Frontier \cup Children$

    return failure



(i): Select $(\pi,s) \in Frontier$ that has largest length(π), i.e., largest number of edges

- Possible tie-breaking rules:
  - left-to-right
  - select smallest $h(s)$ - will discuss later

(ii): Do cycle-checking, then prune all nodes that recursive depth-first search would discard

- Repeatedly remove from *Expanded* any node that has no children in $Children \cup Frontier \cup Expanded$

- Properties
  - Terminates
  - Returns solution if there is one
    - No guarantees on quality
  - Worst-case running time $O(b^l)$
  - Worst-case memory $O(bl)$
    - b = max branching factor
    - l = max depth of any node

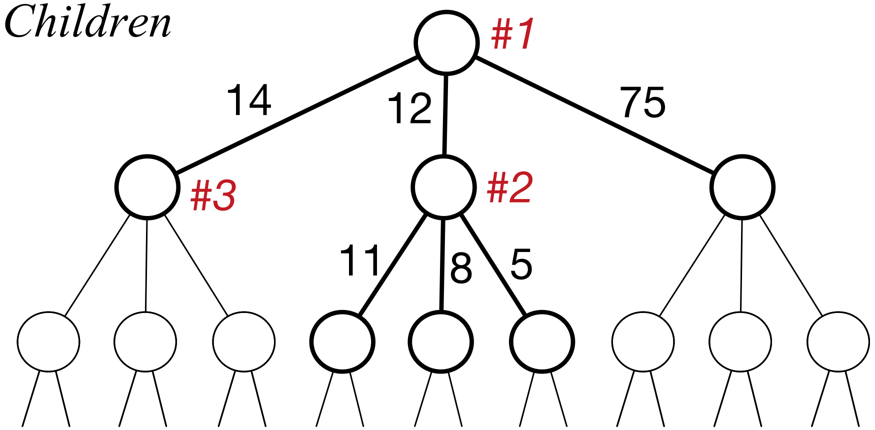
Uniform-Cost Search

Forward-Search-Det(Σ, s_0, g)
    Frontier ← {(⟨⟩, s_0)}
    Expanded ← ∅
    while *Frontier* ≠ ∅ **do**
(*i*)   select a node ν = (π, s) ∈ *Frontier*
        remove ν from *Frontier*
        add ν to *Expanded*
        if *s* satisfies *g* **then return** *π*
        Children ← {($\pi \cdot a$, $\gamma(s,a)$) | a ∈ *Applicable*(s)}
(*ii*)   prune 0 or more nodes from
            Children, *Frontier*, *Expanded*
        Frontier ← *Frontier* ∪ *Children*
    return failure

(*i*): Select (π,s) ∈ *Frontier* that has smallest cost(π)

(*ii*): Prune every (π,s) ∈ *Children* ∪ *Frontier* such that *Expanded* already contains a node (π',s)

- Properties
  - Terminates
  - Finds optimal (i.e., least-cost) solution if one exists
  - Worst-case time $O(b|S|)$
  - Worst-case memory $O(|S|)$

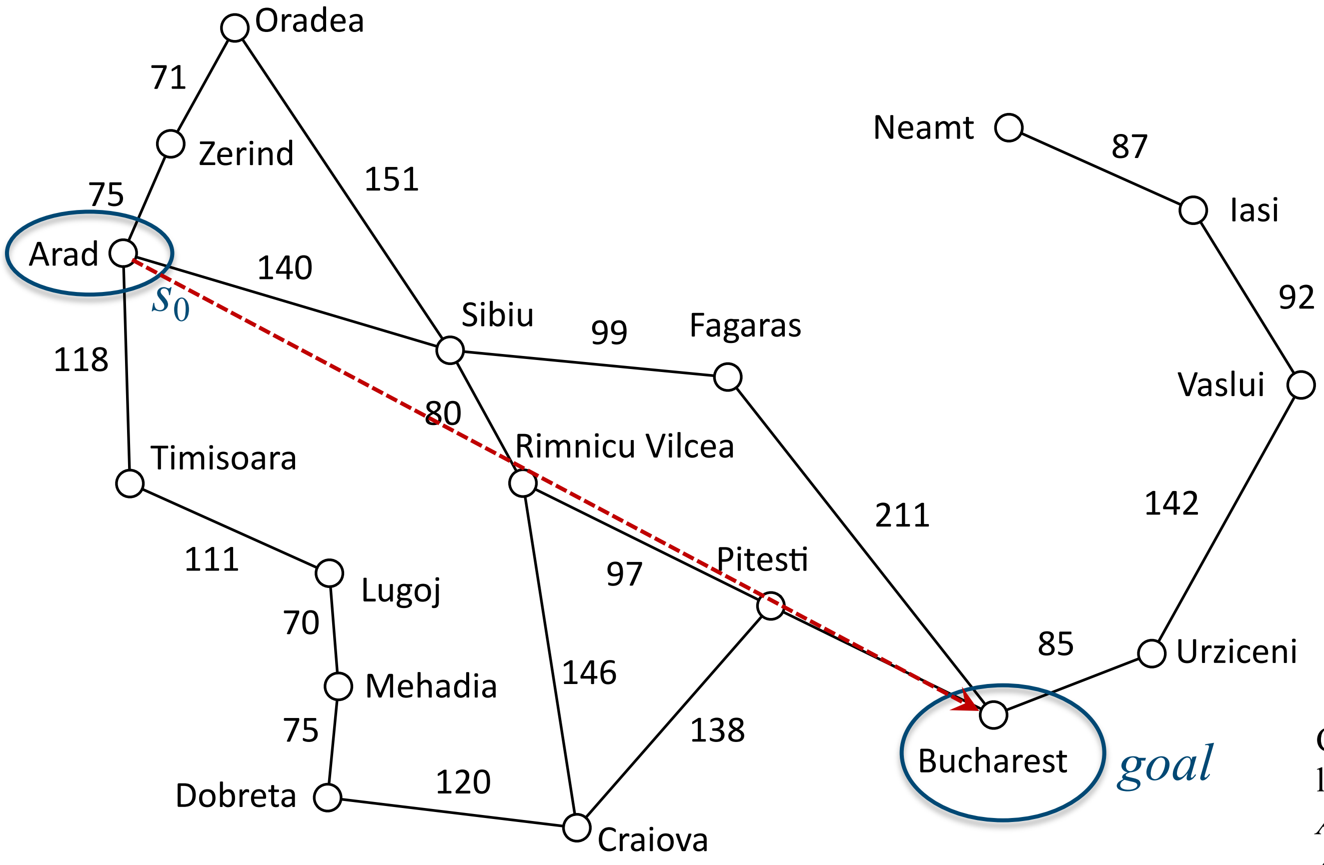


Poll: If node ν is expanded before node ν′, then how are cost(ν) and cost(ν′) related?

A. cost(ν) < cost(ν′)
B. cost(ν) ≤ cost(ν′)
C. cost(ν) > cost(ν′)
D. cost(ν) ≥ cost(ν′)
E. none of the above

Heuristic Functions (more about this later)

- Let $h^*(s)$ = minimum cost of getting to a goal
  $= \min\{\text{cost}(\pi) \mid \gamma(s,\pi) \in S_g\}$
  - Note that $h^*(s) \geq 0$ for all s
- *heuristic function* $h(s)$:
  - Returns estimate of $h^*(s)$
  - Require $h(s) \geq 0$ for all s
- Example:
  - s = the city you're in
  - Action: follow road from s to a neighboring city
  - $h^*(s)$ = smallest distance to Bucharest using roads
  - $h(s)$ = straight-line distance from s to Bucharest



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

Credit: Stuart Russell, lecture slides for *Artificial Intelligence: A Modern Approach*

Greedy Best-First Search (GBFS)

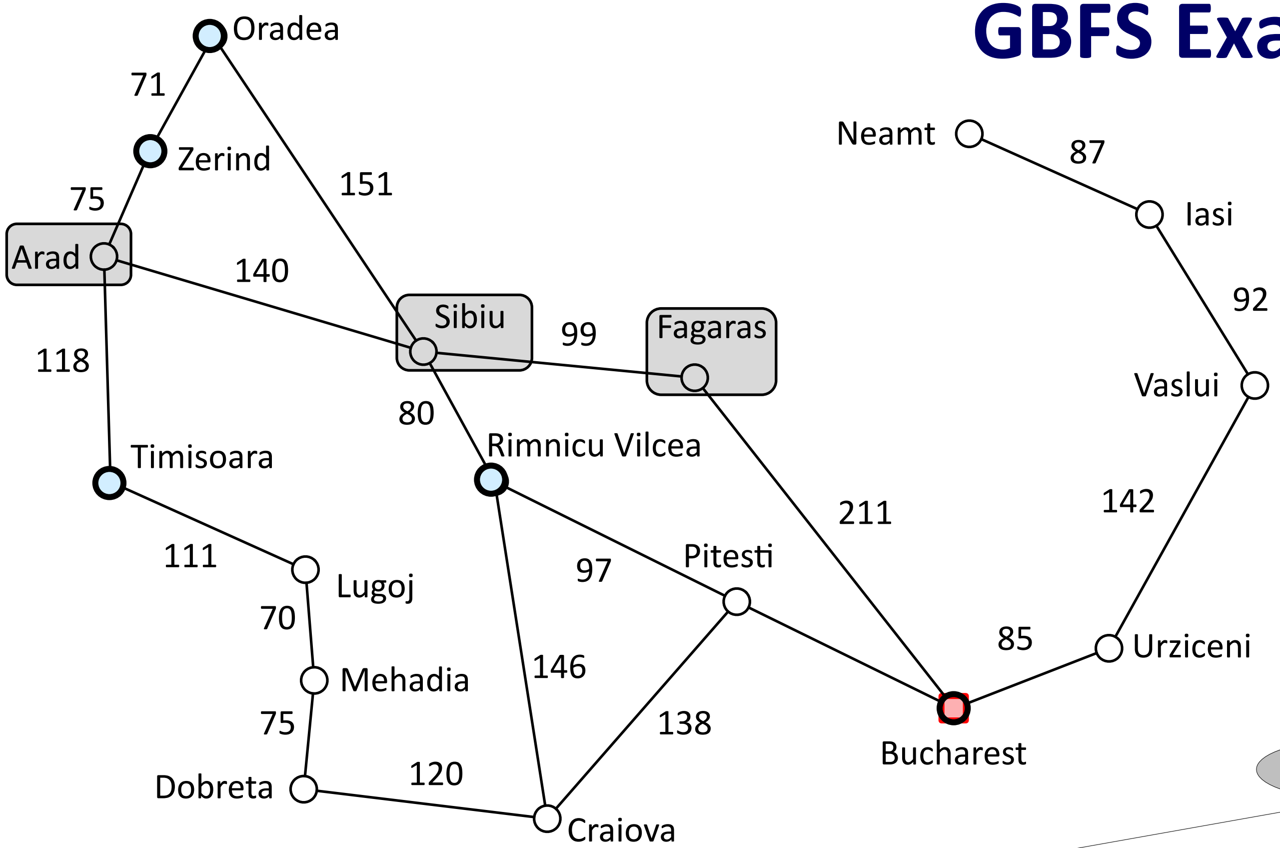
Forward-Search-Det(Σ, s_0, g)
  Frontier ← {(⟨⟩, s_0)}
  Expanded ← ∅
  while *Frontier* ≠ ∅ **do**
(*i*)   select a node ν = (π, s) ∈ *Frontier*
    remove ν from *Frontier*
    add ν to *Expanded*
    if s satisfies g **then return** π
    Children ← $\{(\pi \cdot a, \gamma(s,a)) \mid a \in Applicable(s)\}$
(*ii*)   prune 0 or more nodes from
      Children, *Frontier*, *Expanded*
    Frontier ← *Frontier* ∪ *Children*
  return failure

Poll: Have you seen GBFS before?

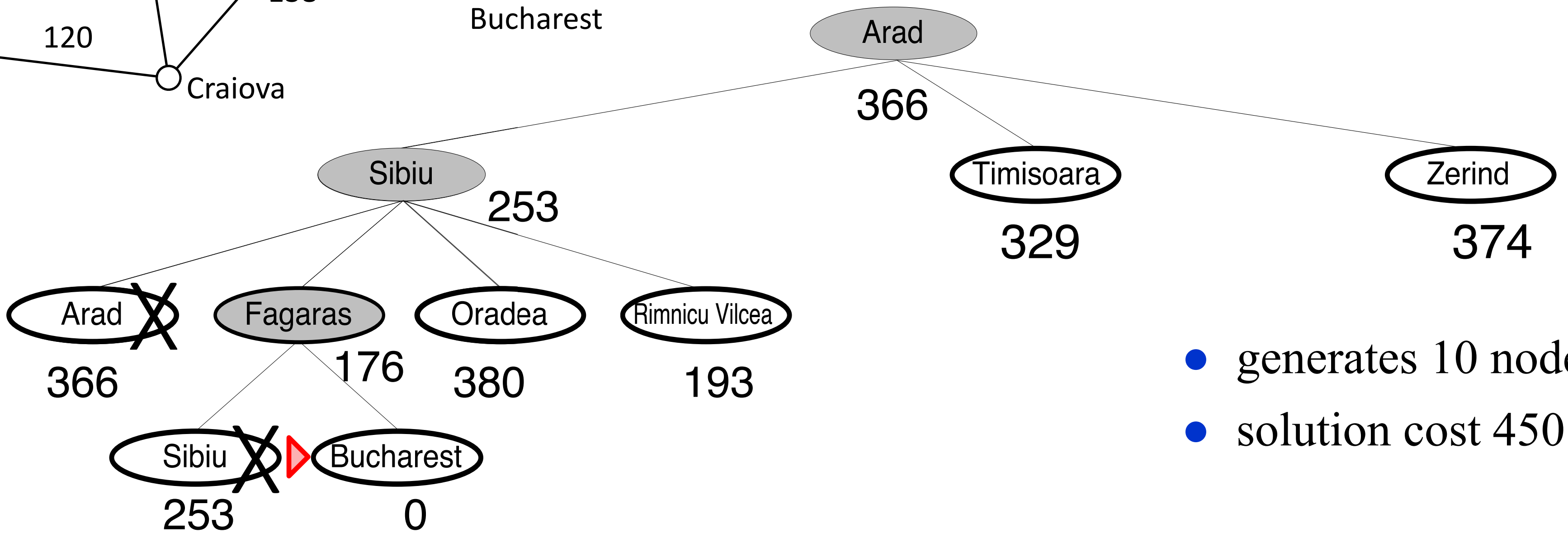
A. yes
B. no
C. yes, but I don't remember it very well

- Idea: choose a node that's likely to be close to a goal
- Node selection:
  - Select a node ν = (π, s) ∈ *Frontier* for which $h(s)$ is smallest
    - Possible tie-breaking rule: choose oldest
- Pruning: should at least include cycle checking.
  - For other cases where two nodes go to the same state s, several possibilities:
    - Prune one of the nodes arbitrarily
    - Prune the higher-cost node
    - Do no pruning (with a good heuristic function, GBFS is unlikely to expand both nodes)
- Properties
  - Terminates; returns a solution if one exists
  - Solution is usually found quickly, often near-optimal

GBFS Example



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



- generates 10 nodes
- solution cost 450

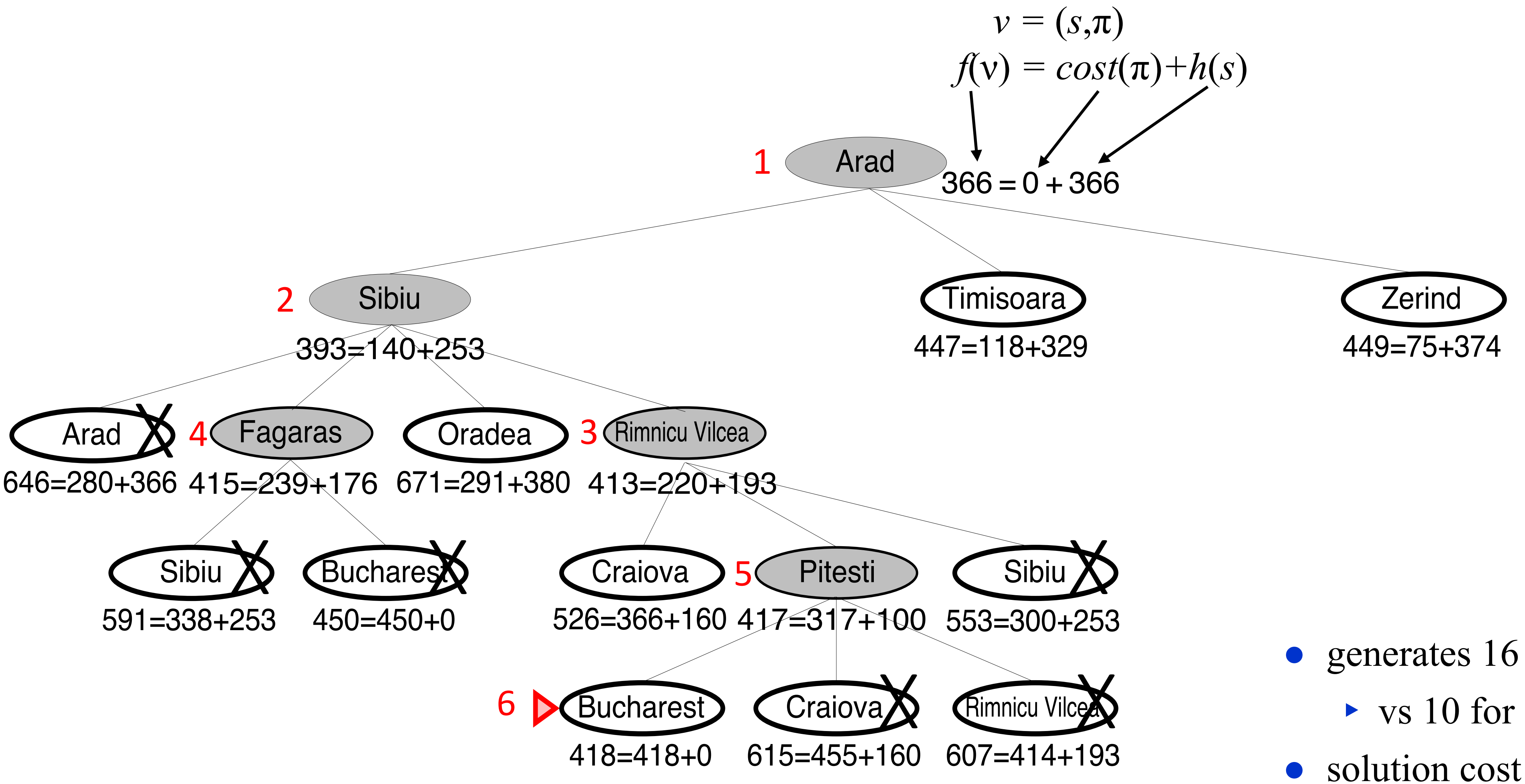
A*

Forward-Search-Det(Σ, s_0, g)
    Frontier ← $\{(\langle\rangle, s_0)\}$
    Expanded ← ∅
    while *Frontier* ≠ ∅ **do**
(*i*)   select a node ν = (π, s) ∈ *Frontier*
        remove ν from *Frontier*
        add ν to *Expanded*
        if s satisfies g **then return** π
        Children ← $\{(\pi \cdot a, \gamma(s,a)) \mid a \in \mathit{Applicable}(s)\}$
(*ii*)   prune 0 or more nodes from
            Children, *Frontier*, *Expanded*
        Frontier ← *Frontier* ∪ *Children*
    return failure

Poll: Have you seen A* before?
A. yes
B. no
C. yes, but I don't remember it very well

- Idea: try to choose a node on an optimal path from s_0 to goal
- Node selection
  - Select a node ν = (π,s) in *Frontier* that has smallest value of $f(\nu) = \mathrm{cost}(\pi) + h(s)$
    - Possible tie-breaking rule: select oldest
- Pruning:
  - for every node ν = (π,s) in *Children*:
    - If *Children* ∪ *Frontier* ∪ *Expanded* contains another node with the same state s, then we've found multiple paths to s
    - Keep only the one with the lowest cost
    - If more than one such node, keep the oldest
- Properties (in classical planning problems):
  - *Termination*: Always terminates
  - *Complete*: returns a solution if one exists
  - *Optimality*: can guarantee this under certain conditions (I'll discuss later)

A* Example



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

- generates 16 nodes
  - vs 10 for GBFS
- solution cost 418
  - vs 450 for GBFS

Admissibility

- Notation:
  - $\nu = (\pi, s)$, where π is the plan for going from s_0 to s
  - $h^*(s) = \min\{\text{cost}(\pi') \mid \gamma(s,\pi') \text{ satisfies } g\}$
  - $f^*(\nu) = \text{cost}(\pi) + h^*(s)$
  - $f(\nu) = \text{cost}(\pi) + h(s)$

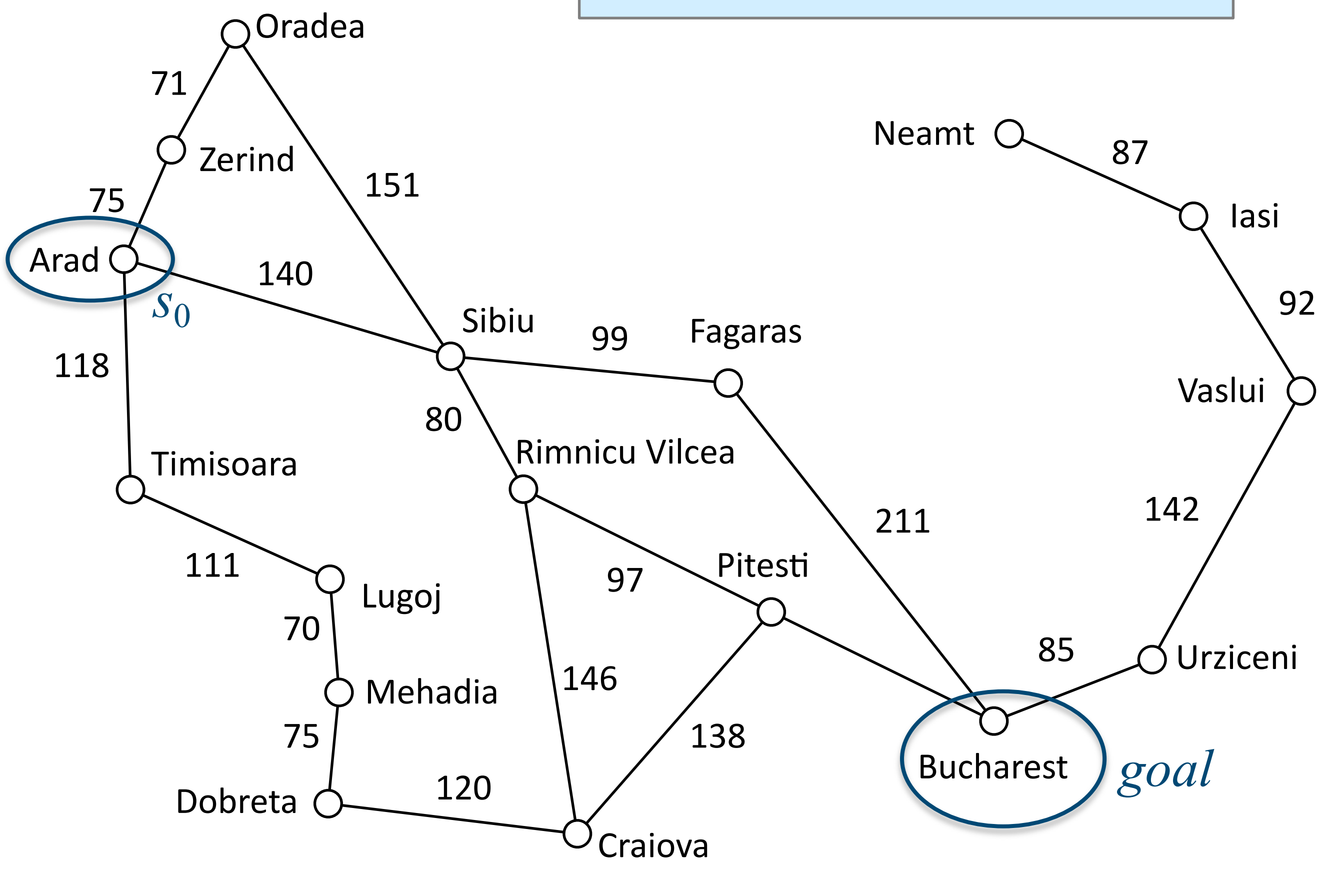
- Definition: h is *admissible* if for every s, $h(s) \leq h^*(s)$

- *Optimality:*
  - if h is admissible then every solution returned by A* will be optimal (least cost)

Poll: If $h(s)$ = straight-line distance from s to Bucharest, is h admissible?

A. Yes B. No C. Not sure

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



Admissibility

- Notation:
  - $v = (\pi, s)$, where π is the plan for going from s_0 to s
  - $h^*(s) = \min\{\text{cost}(\pi') \mid \gamma(s,\pi') \text{ satisfies } g\}$
  - $f^*(v) = \text{cost}(\pi) + h^*(s)$
  - $f(v) = \text{cost}(\pi) + h(s)$

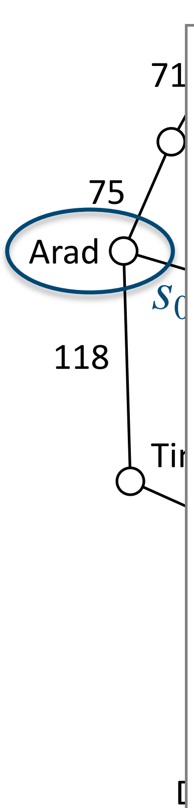
- Definition: h is *admissible* if for every s, $h(s) \leq h^*(s)$

- *Optimality:*
  - if h is admissible then every solution returned by A* will be optimal (least cost)

Poll: If h is admissible, does it follow that for every expanded node v, $f(v) \leq f^*(v)$?

Poll: If h is admissible, does it follow that for every node v, $f(v) \leq f^*(v)$?

A. Yes B. No C. Not sure



Forward-Search-Det(Σ, s_0, g)
  Frontier ← $\{(\langle\rangle, s_0)\}$
  Expanded ← ∅
  while *Frontier* ≠ ∅ **do**
    select a node $v = (\pi, s) \in$ *Frontier*
    remove v from *Frontier*
    add v to *Expanded*
    if s satisfies g **then return** π
    Children ← $\{(\pi{\cdot}a, \gamma(s,a)) \mid a \in \textit{Applicable}(s)\}$
    prune 0 or more nodes from
      Children, *Frontier*, *Expanded*
    Frontier ← *Frontier* ∪ *Children*
  return failure

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

Dominance

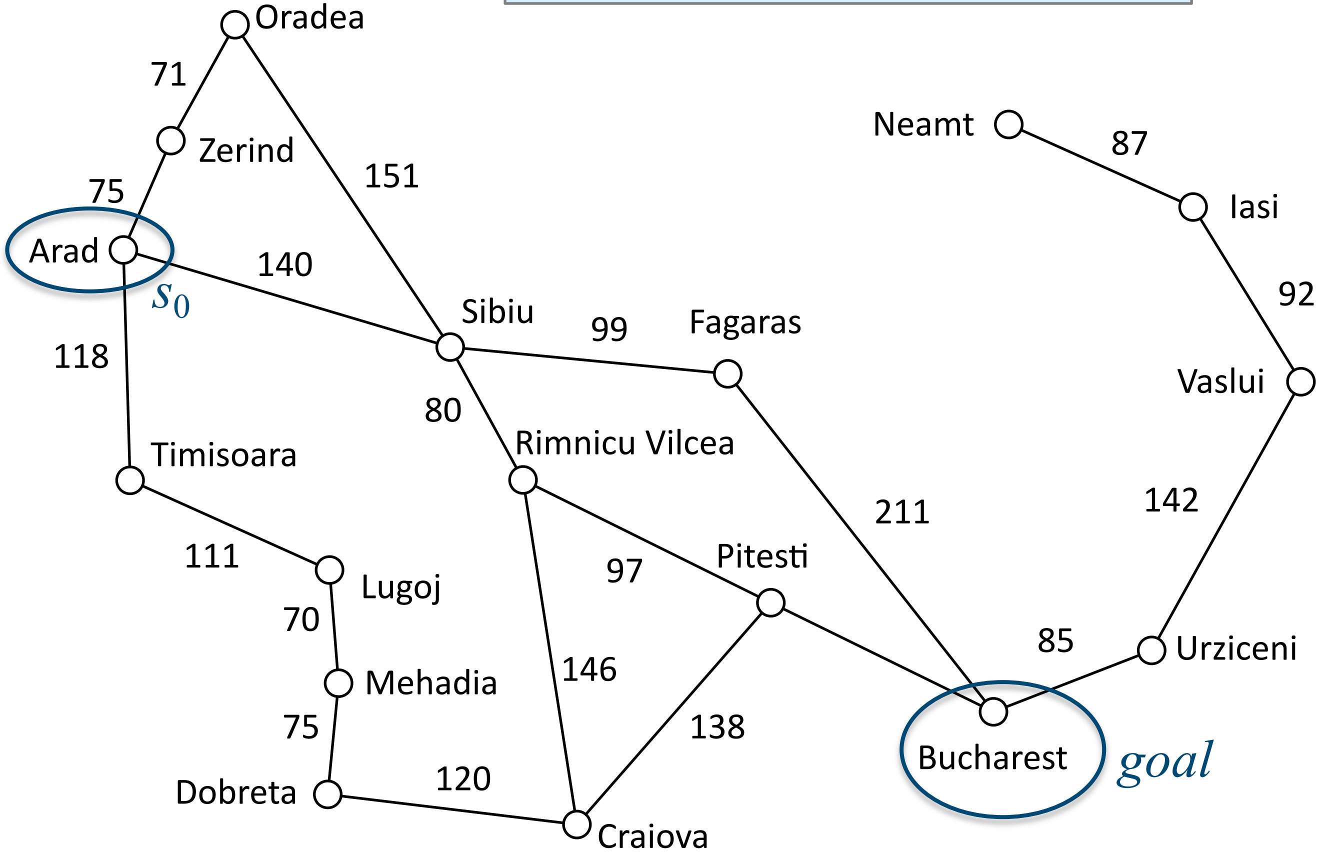
- Definition:
  - Let h_1, h_2 be admissible heuristic functions
  - h_2 *dominates* h_1 if $\forall s$, $h_1(s) \le h_2(s) \le h^*(s)$

Poll: Let $h_1(s) = 0$ and $h_2(s)$ = straight-line distance from s to Bucharest. Does h_2 dominate h_1?

A. Yes B. No C. Not sure

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

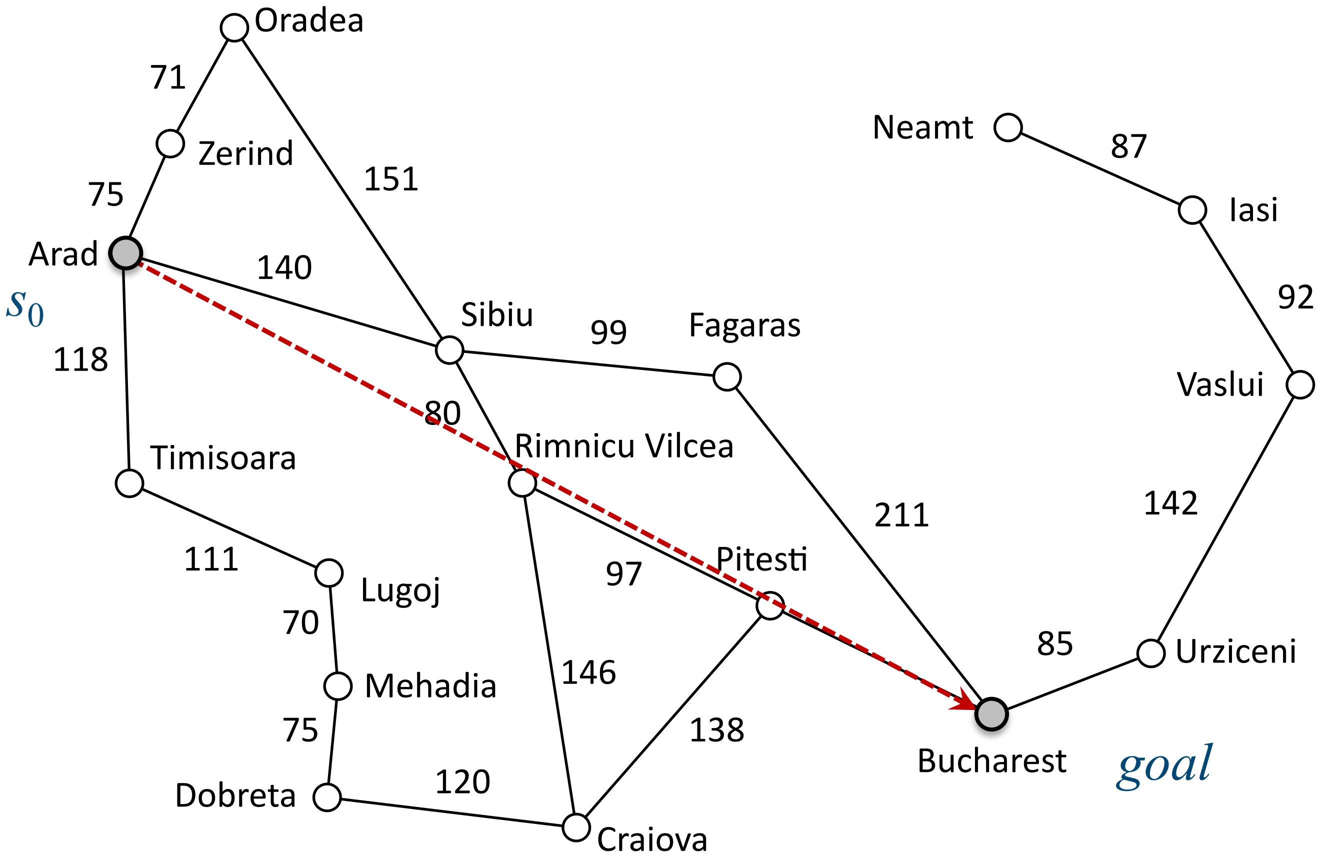
- Suppose h_2 dominates h_1, and A* always resolves ties in favor of the same node. Then
  - A* with h_2 will never expand more nodes than A* with h_1
  - In most cases, A* with h_2 will expand fewer nodes than A* with h_1



Digression

- Straight-line distance to Bucharest is a *domain-specific* heuristic function
  - OK for planning a path to Bucharest
  - Not for other planning problems

- *Domain-independent* heuristic function:
  - A heuristic function that can be used in any classical planning domain
  - Many such heuristics (see Section 3.2)



Properties of A*

In classical planning problems:

- *Termination:* A* will always terminate
- *Completeness:* if the problem is solvable, A* will return a solution
- *Optimality:* if h is admissible then the solution will be optimal (least cost)
- *Dominance:* If h_2 dominates h_1 and if A* always resolves ties the same way
  - A* with h_2 will never expand more nodes than A* with h_1
  - In most cases, A* with h_2 will expand fewer nodes than A* with h_1

- A* needs to store every node it visits
  - Running time $O(b|S|)$ and memory $O(|S|)$ in worst case
  - With good heuristic function, usually much smaller

- The book discusses additional properties

Comparison

- If h is admissible, A* will return optimal solutions
  - But running time and memory requirement grow exponentially in b and d

- GBFS returns the first solution it finds
  - There are cases where GBFS takes more time and memory than A*
    - But with a good heuristic function, such cases are rare
  - On classical planning problems with a good heuristic function
    - GBFS usually near-optimal solutions
    - GBFS does very little backtracking
    - Running time and memory requirement usually much less than A*

  - GBFS is used by most classical planners nowadays

Depth-First Branch and Bound (DFBB)

- Basic idea:
  - depth-first search, but don't stop at the first solution
  - π^* = best solution so far
  - $c^* = \text{cost}(\pi^*)$
  - prune ν if $f(\nu) \geq c^*$
  - when frontier is empty, return π^*
- Properties
  - Termination, completeness, optimality same as A*
  - Usually less memory, more time than A*
  - Worst-case like DFS:
    - $O(bl)$ memory, $O(b^l)$ time

Forward-Search-Det(Σ, s_0, g)
    Frontier ← $\{(\langle\rangle, s_0)\}$
    Expanded ← ∅
    $c^* \leftarrow \infty$; $\pi^* \leftarrow$ failure
    while *Frontier* ≠ ∅ **do**
(*i*)   select a node $\nu = (\pi, s) \in$ *Frontier*
        remove ν from *Frontier*
        add ν to *Expanded*
        ~~**if** *s* satisfies *g* **then return** π~~
        if *s* satisfies *g* and $\text{cost}(\pi) < c^*$ then
            $c^* \leftarrow \text{cost}(\pi)$; $\pi^* \leftarrow \pi$
(*ii*)   else if $f(\nu) < c^*$ then
            Children ←
              $\{(\pi \cdot a, \gamma(s,a)) \mid a \in Applicable(s)\}$
(*iii*)   prune 0 or more nodes from
              Children, *Frontier*, *Expanded*
            Frontier ← *Frontier* ∪ *Children*
    return ~~failure~~ π^*

Poll: Have you seen DFBB before?

A. yes   B. no

C. yes, but don't remember it very well

- Can write it as a modified version of Forward-Search-Det
- Node selection:
  - (*i*) same as in DFS
- Pruning:
  - (*ii*) If $f(\nu) \geq c^*$ then discard
  - (*iii*) Otherwise prune the same nodes as in DFS
- Don't stop until every node has been visited or pruned

Comparisons

- If h is admissible, both A* and DFBB will return optimal solutions
  - Usually DFBB generates more nodes, but A* takes more memory
  - Worst case for DFBB:
    - Highly connected graphs (many paths to each state)
    - Can have exponentially worse running time than A* (generates nodes exponentially many times)
  - Best case for DFBB:
    - Search space is a tree of uniform height, all solutions at the bottom (e.g., constraint satisfaction)
    - DFBB and A* have similar running time
    - A* can take exponentially more memory than DFBB

- DFS returns the first solution it finds
  - can take much less time than DFBB
  - but solution can be very far from optimal

Iterative Deepening (IDS)

IDS(Σ, s_0, g)

for $k = 1$ to ∞ **do**

do a depth-first search, backtracking at every node of depth k

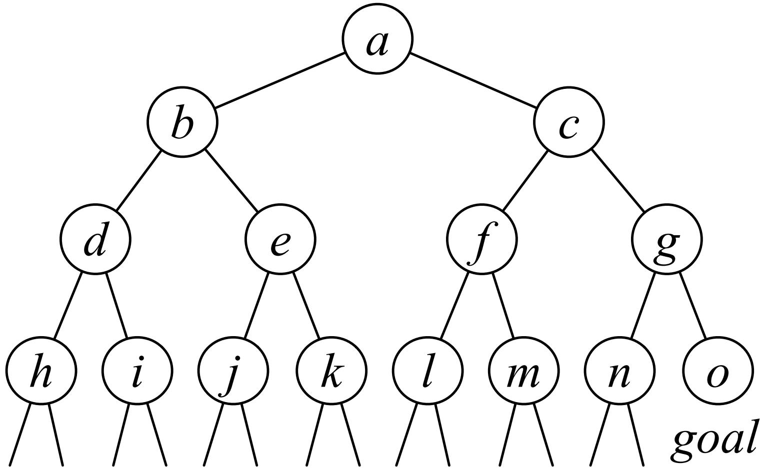
if the search found a solution **then** return it

if the search generated no nodes of depth k **then** return failure

Poll: Have you seen Iterative Deepening before?

A. yes
B. no
C. yes, but I don't remember it very well

- Nodes generated:
  - a,b,c
  - a,b,c,d,e,f,g
  - $a,b,c,d,e,f,g,h,i,j,k,l,m,n,o$



- Solution path $\langle a,c,g,o \rangle$
- Total number of nodes generated:

  $3+7+15 = 25$

- If goal is at depth d and branching factor is 2:
  - $\sum_1^d (2^{i+1}-1) = \sum_1^d 2^{i+1} - \sum_1^d 1 = O(2^d)$

Poll: How many nodes generated if branching factor is b instead of 2?

A. $O(b2^d)$
B. $O((b/2)^d)$
C. $O(b^d)$
D. $O(b^{d+1})$
E. something else

Iterative Deepening (IDS)

IDS(Σ, s_0, g)

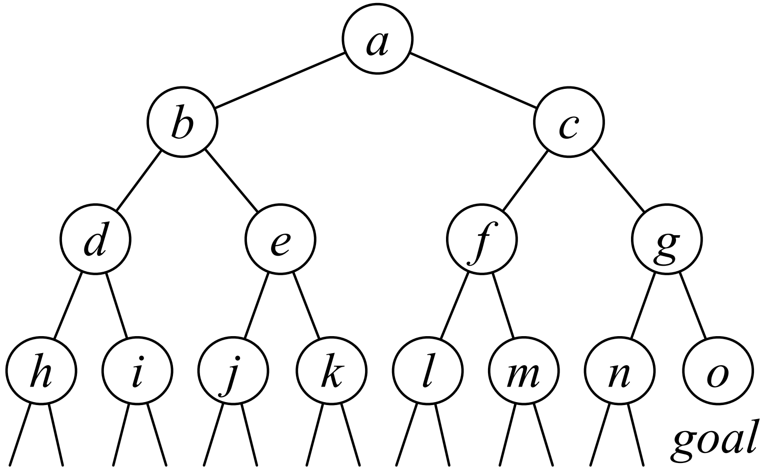
  for $k = 1$ to ∞ **do**

    do a depth-first search, backtracking at every node of depth k

    if the search found a solution **then** return it

    if the search generated no nodes of depth k **then** return failure

- Nodes generated:
  - a,b,c
  - a,b,c,d,e,f,g
  - $a,b,c,d,e,f,g,h,i,j,k,l,m,n,o$
- Solution path $\langle a,c,g,o \rangle$
- Total number of nodes generated:
  - $3+7+15 = 25$
- If goal is at depth d and branching factor is 2:
  - $\sum_1^d (2^{i+1}-1) = \sum_1^d 2^{i+1} - \sum_1^d 1 = O(2^d)$



Properties:

- Termination, completeness, optimality
  - same as BFS
- Memory (worst case): $O(bd)$
  - vs. $O(b^d)$ for BFS
- If the number of nodes grows exponentially with d:
  - worst-case running time $O(b^d)$, vs. $O(b^l)$ for DFS
  - b = max branching factor
  - l = max depth of any node
  - d = min solution depth if there is one, otherwise l

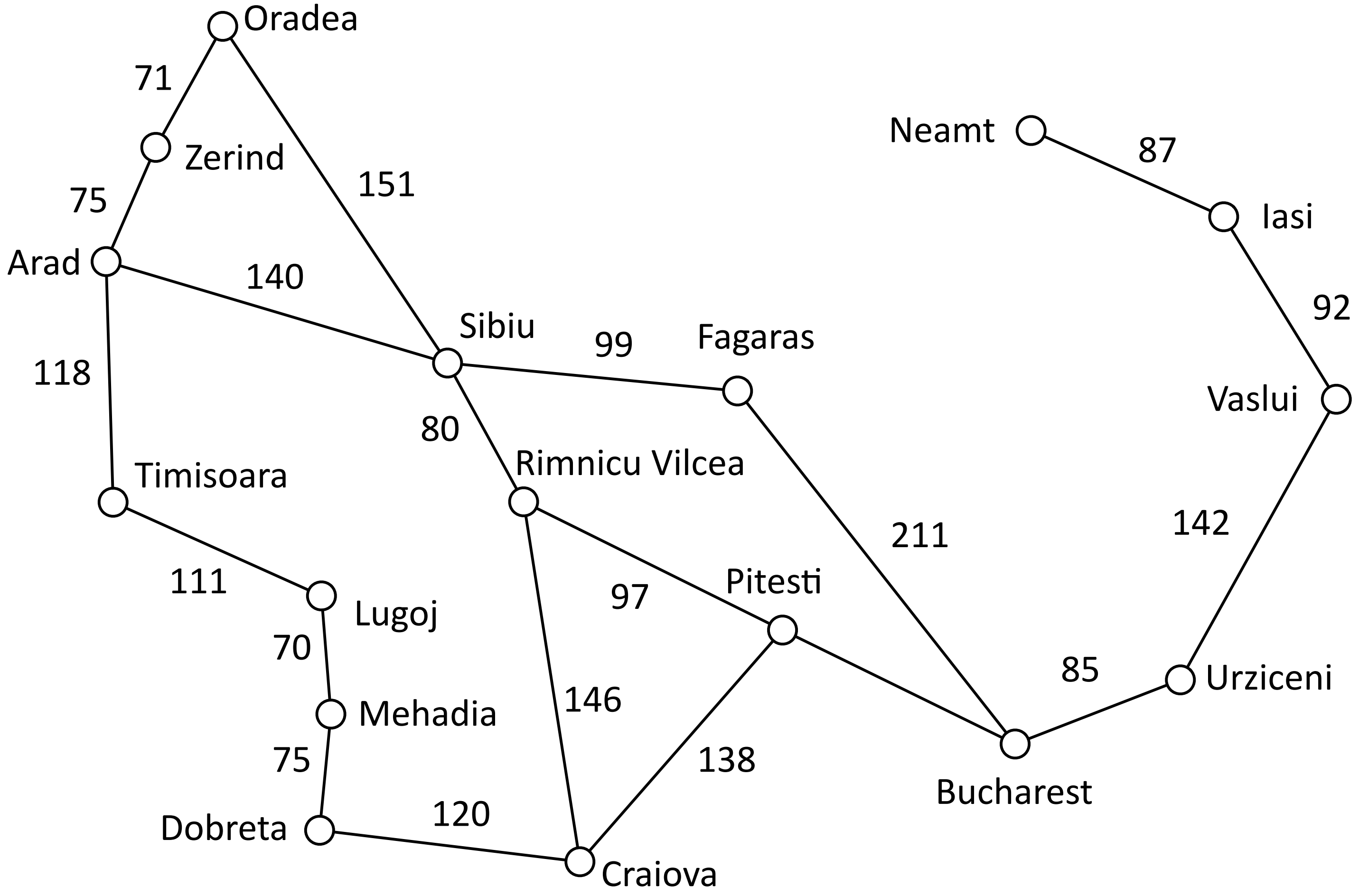
3.2. Heuristic Functions

- 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

Example

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

    $\Rightarrow$ 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

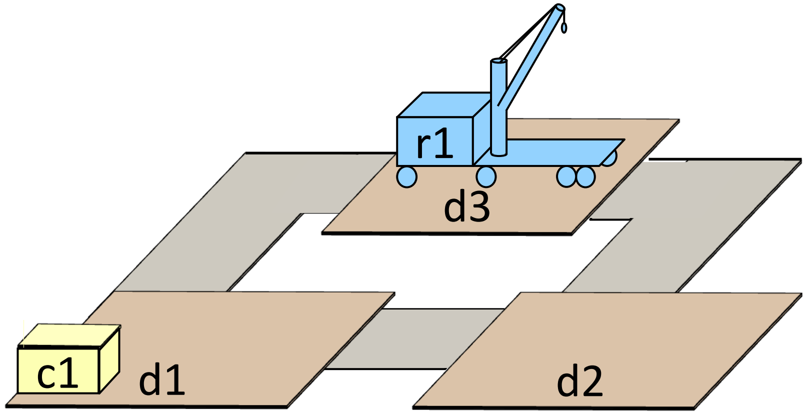
Domain-independent Heuristics

- Use relaxation to get heuristic functions that can be used in any classical planning problem
  - Delete-relaxation heuristics
    - Optimal relaxed solution
    - Fast-Forward heuristic
  - Landmark heuristics
  - Max-cost and additive-cost heuristics (I'll skip these)

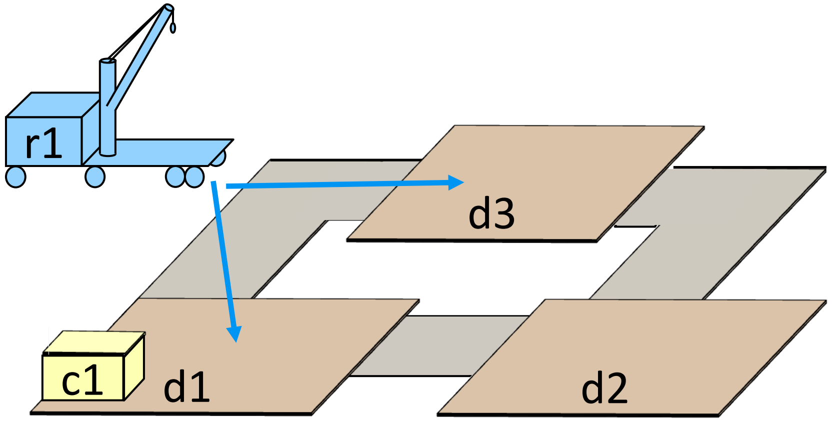
3.2.1. Delete-Relaxation

- 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 this 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

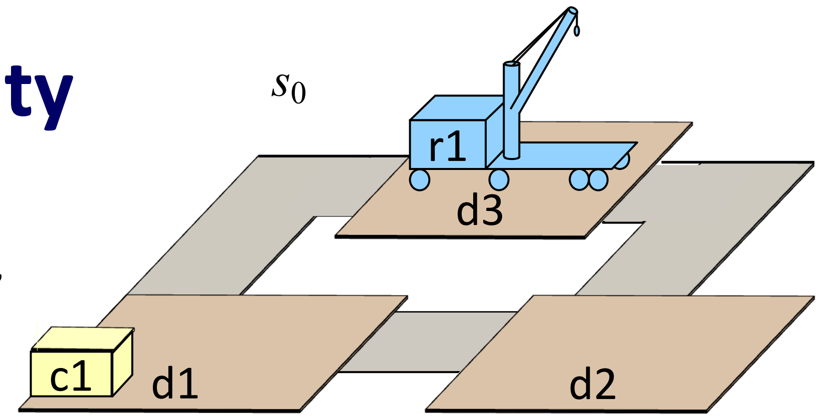
- Action a is *r-applicable* in a relaxed state $\hat{s}$ if an *r-subset* of $\hat{s}$ satisfies a's preconditions
  - a subset with one value per state variable
- If a is r-applicable then $\gamma^{+}(\hat{s},a) = \hat{s} \cup \gamma(s,a)$

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

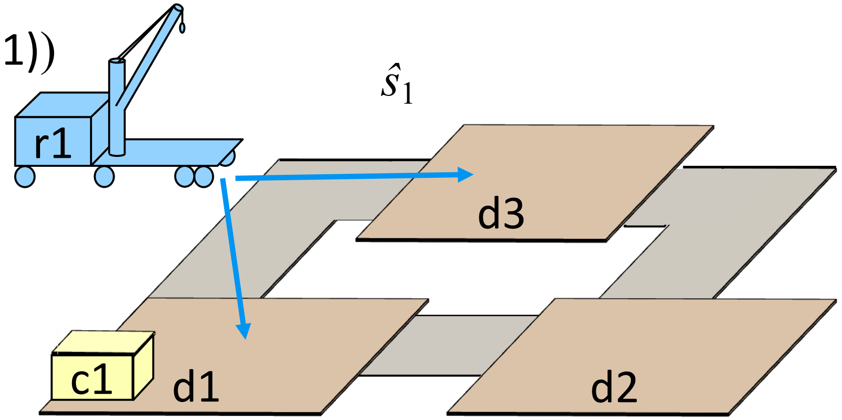
move(r, d, e)
  pre: loc(r)=d
  eff: loc(r)←e

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

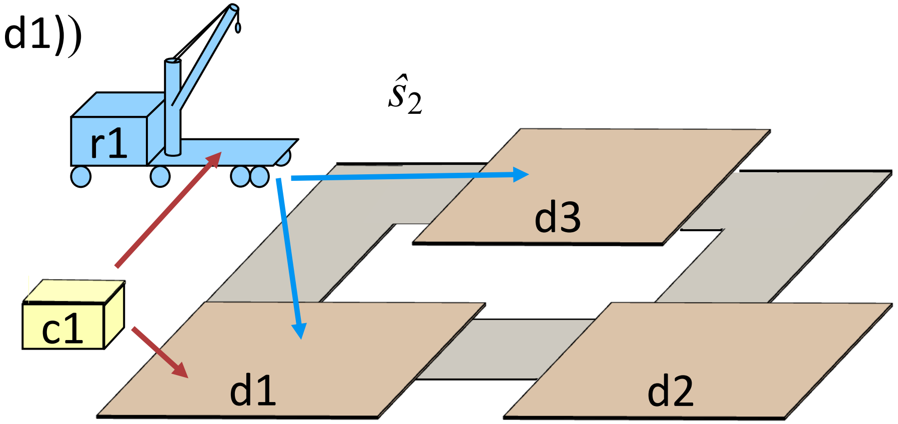
s_0 = {loc(r1) = d3,
cargo(r1) = nil,
loc(c1) = d1}



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

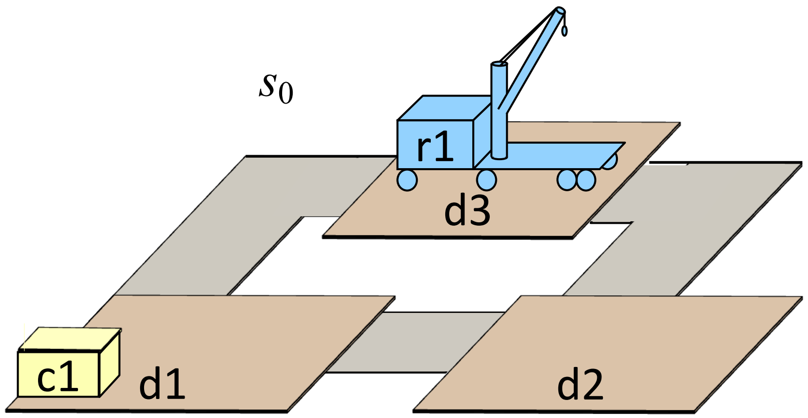


Relaxed Applicability (continued)

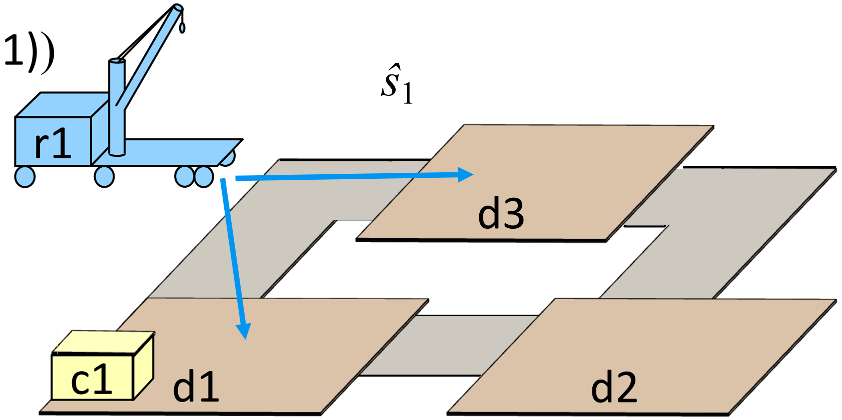
- Let $\pi = \langle a_1, \ldots, a_n \rangle$ be a plan
- Suppose we can r-apply the actions of π in the order $a_1, \ldots, a_n$:
  - r-apply a_1 in $\hat{s}_0$, get $\hat{s}_1 = \gamma^+(\hat{s}_0, a_1)$
  - r-apply a_2 in $\hat{s}_1$, get $\hat{s}_2 = \gamma^+(\hat{s}_1, a_2)$
  - …
  - r-apply a_n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

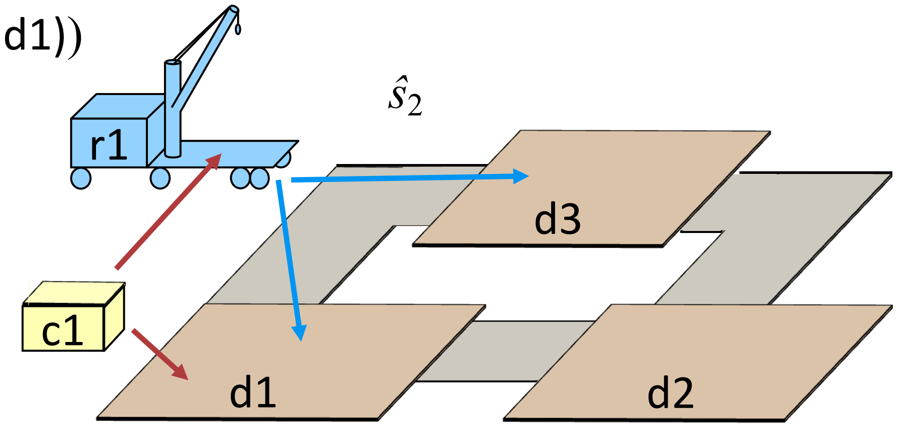
s_0 = {loc(r1) = d3,
cargo(r1) = nil,
loc(c1) = d1}



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

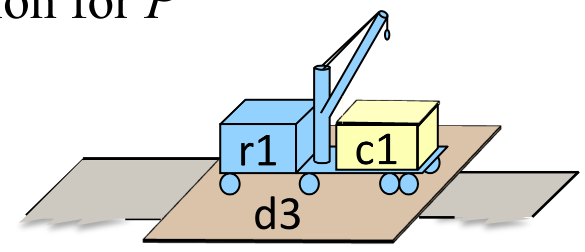


$\hat{s}_2 = \gamma^+(\hat{s}_1,$ load(r1,c1,d1))
= {loc(r1) = d1,
loc(r1) = d3,
cargo(r1) = nil,
cargo(r1) = c1,
loc(c1) = r1,
loc(c1) = d1}



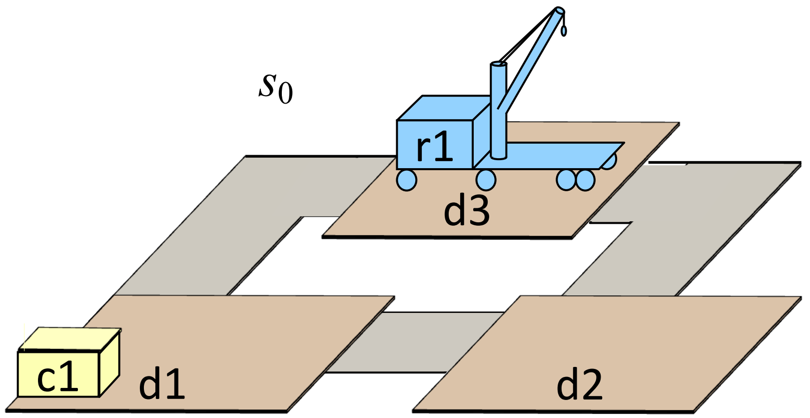
Relaxed Solution

- An r-state $\hat{s}$ *r-satisfies* a formula g if an r-subset of $\hat{s}$ satisfies g
  - a subset with one value per state variable
- *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

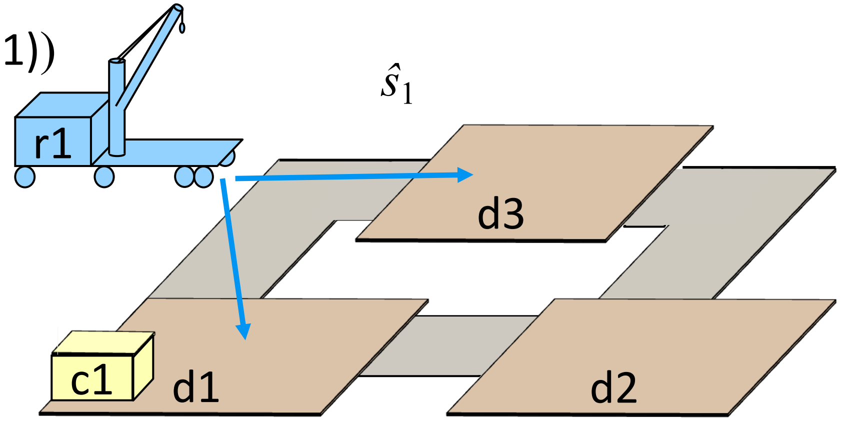


g = {loc(r1)=d3, loc(c1)=r1}

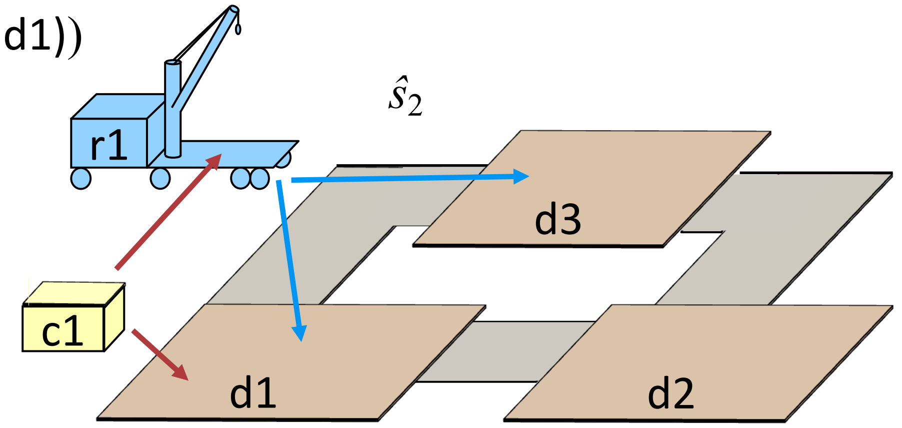
s_0 = {loc(r1) = d3,
cargo(r1) = nil,
loc(c1) = d1}



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



Relaxed Solution

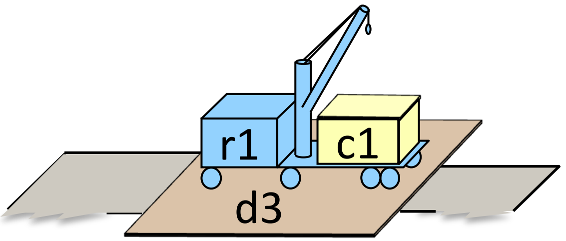
- Planning problem $P = (\Sigma, s_0, g)$
- *Optimal relaxed solution* heuristic:
  - $h^+(s)$ = minimum cost of all relaxed solutions for (Σ, s, g)

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

Poll: is h^+ admissible?

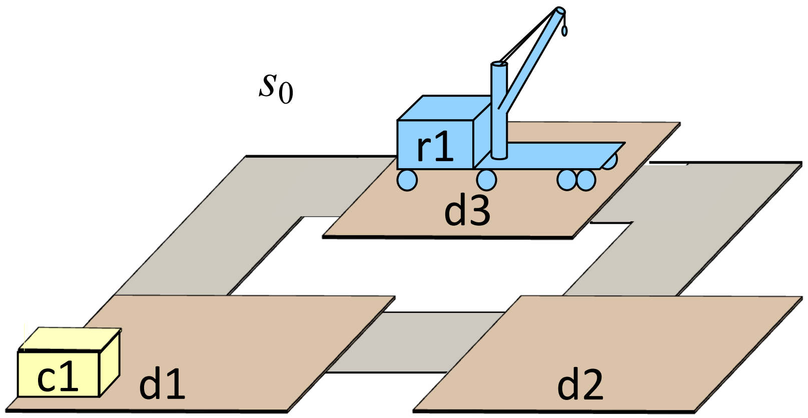
A. Yes

B. No

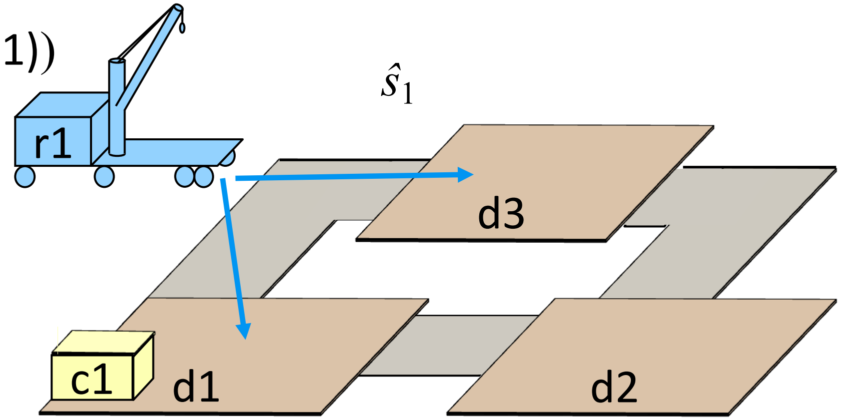


g = {loc(r1)=d3, loc(c1)=r1}

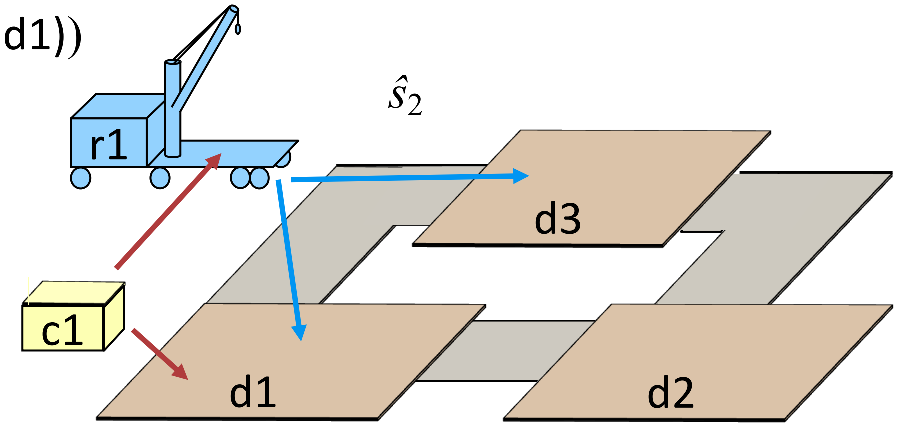
s_0 = {loc(r1) = d3,
cargo(r1) = nil,
loc(c1) = d1}



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



Example: GBFS

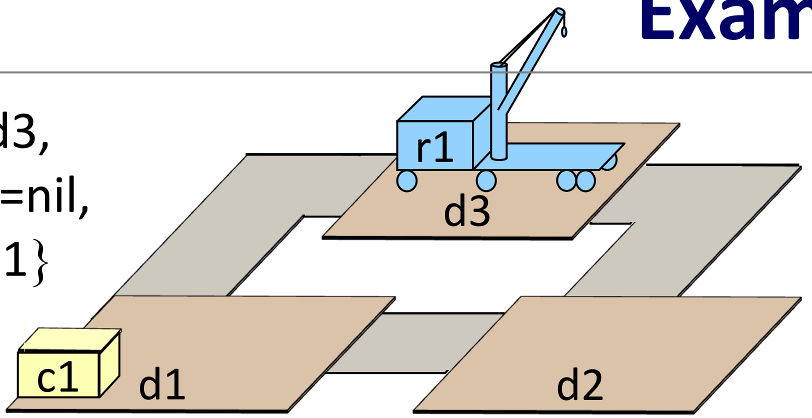
Poll 1: What is $h^{+}(s_1)$?

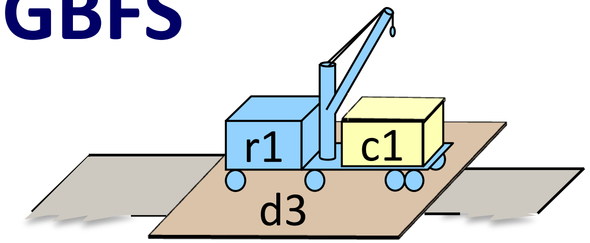
A. 1
B. 2
C. 3
D. 4
E. other

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$\}$



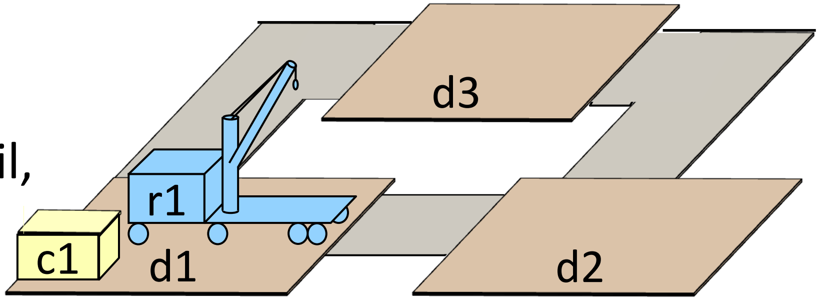


$g = \{$loc(r1)=d3, loc(c1)=r1$\}$

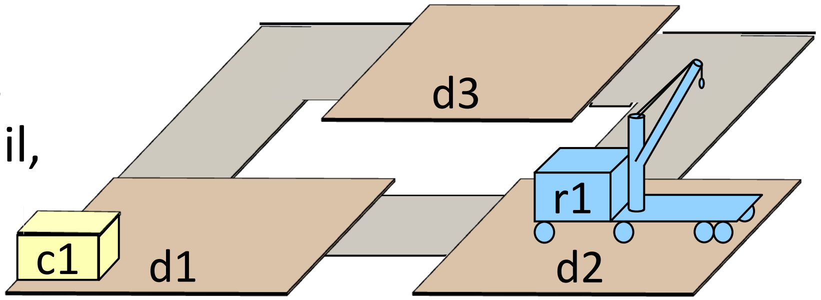
$a_1 =$ move(r1,d3,d1)

$a_2 =$ 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 $h^+(s)$ 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

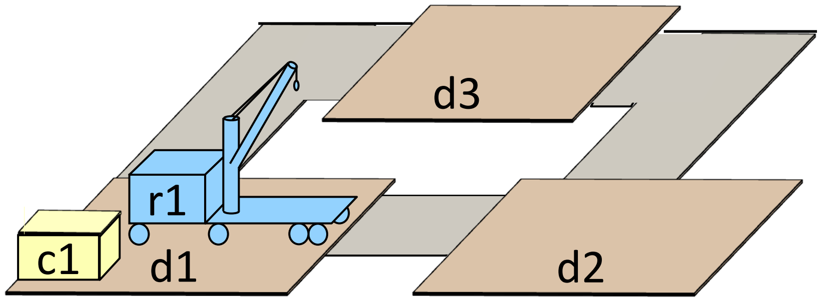
Preliminaries

- 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 $\Rightarrow$ same state $\hat{s}_2$

- Let A_1 be a set of actions that all are r-applicable in $\hat{s}_0$
  - 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 $\text{eff}(A_1) = \bigcup\{\text{eff}(a) \mid a \in A_1\}$
- Suppose A_2 is a set of actions that are r-applicable in $\hat{s}_1$
  - $\hat{s}_1$ satisfies $\text{pre}(A_2) = \bigcup\{\text{pre}(a) \mid a \in A_2\}$
  - $\hat{s}_2 = \gamma^+(\hat{s}_0, \langle A_1, A_2\rangle) = \hat{s}_0 \cup \text{eff}(A_1) \cup \text{eff}(A_2)$

…

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



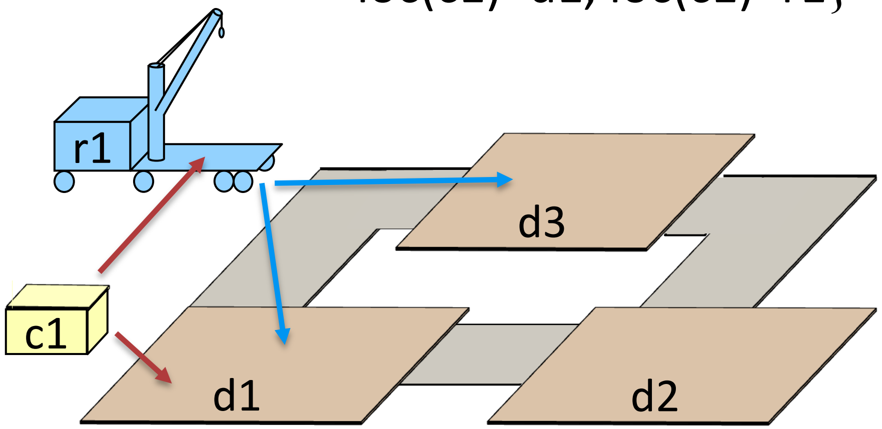
s_0 = {loc(r1)=d1, cargo(r1)=nil, loc(c1)=d1}

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

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^+

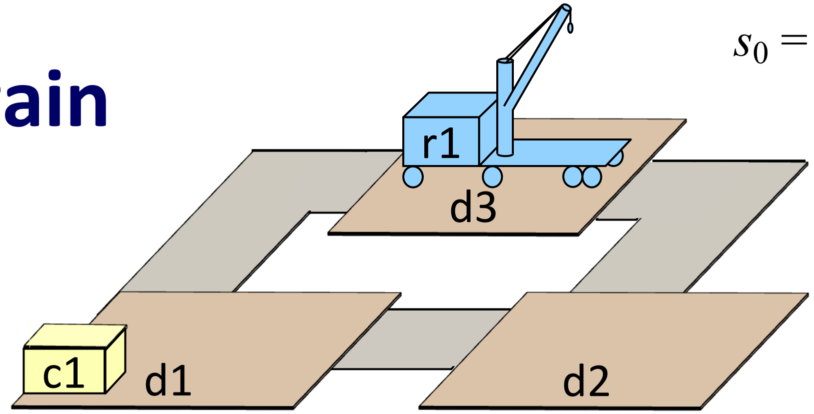
$\text{pre}(\hat{a}_i) = \bigcup \{\text{pre}(a) \mid a \in \hat{a}_i\}$
$\text{eff}(\hat{a}_i) = \bigcup \{\text{eff}(a) \mid a \in \hat{a}_i\}$

ambiguous →

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

Example: GBFS Again

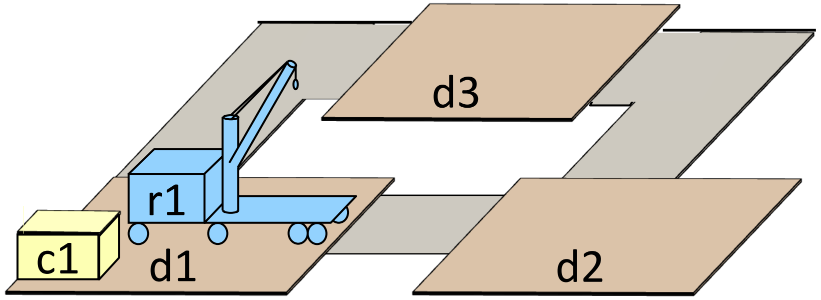
- GBFS with initial state s_0, goal g, heuristic h^{FF}
- Two applicable actions: a_1, a_2
- Resulting states: s_1, s_2
- GBFS computes $h^{\mathrm{FF}}(s_1)$ and $h^{\mathrm{FF}}(s_2)$
  - Chooses the state that has the lower h^{FF} value
- Next several slides:
  - $h^{\mathrm{FF}}(s_1)$
  - $h^{\mathrm{FF}}(s_2)$



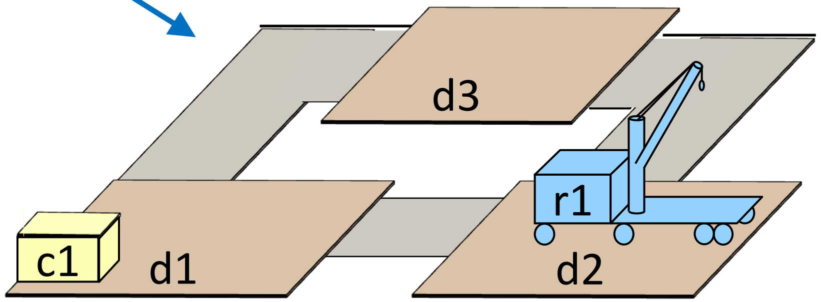
s_0 = {loc(c1) = d1, loc(r1) = d3, cargo(r1) = nil}

a_1 = move(r1,d3,d1)

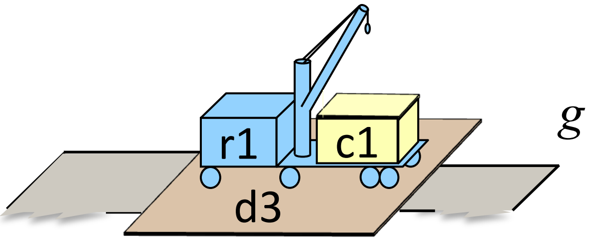
a_2 = move(r1,d3,d2)



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



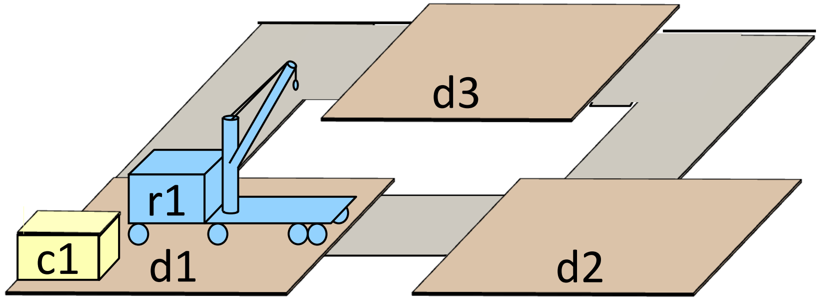
$s_2 = \gamma(s_0,a_2)$ = {loc(c1) = d1, loc(r1) = d2, cargo(r1) = nil}



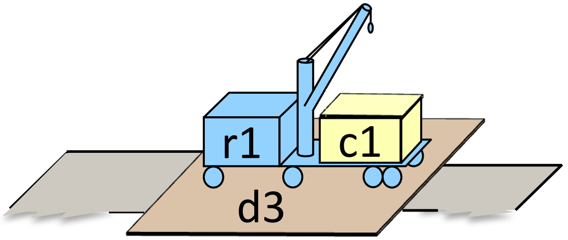
g = {loc(r1)=d3, loc(c1)=r1}

Example

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



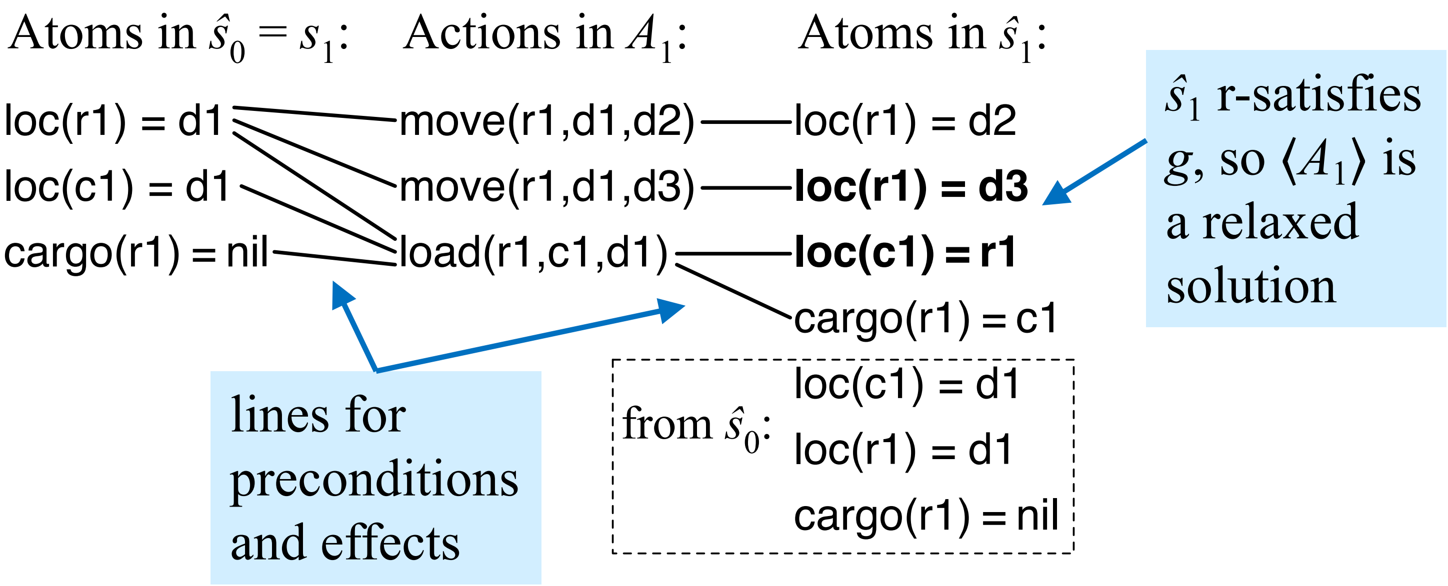
s_1 = {loc(r1)=d1, cargo(r1)=nil, loc(c1)=d1}



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

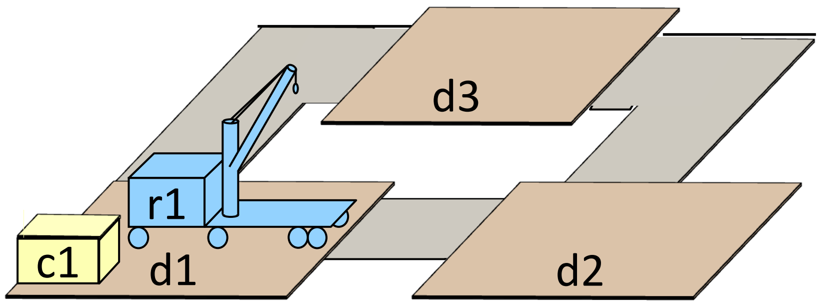
// *construct a relaxed solution* $\langle A_1, A_2, \ldots, A_k \rangle$:
$\hat{s}_0 \leftarrow s$
for $k = 1$ by 1 until $\hat{s}_k$ r-satisfies g
  $A_k \leftarrow$ {all actions r-applicable in $\hat{s}_{k-1}$}; $\hat{s}_k \leftarrow \gamma^+(s_{k-1}, A_k)$
  if $k > 1$ and $\hat{s}_k = \hat{s}_{k-1}$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

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

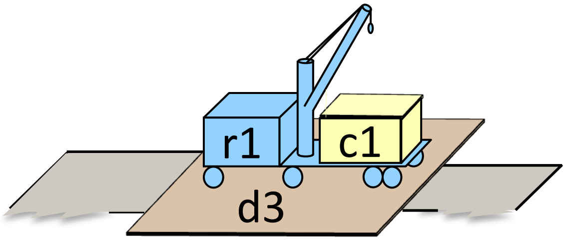


Example

- 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 $\hat{g}_1 = g$

Atoms in $\hat{s}_0 = s_1$:
- **loc(r1) = d1**
- **loc(c1) = d1**
- **cargo(r1) = nil**

($\hat{g}_0$)

Actions in A_1:
- move(r1,d1,d2)
- **move(r1,d1,d3)**
- **load(r1,c1,d1)**

($\hat{a}_1$)

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

($\hat{g}_1 = g$)

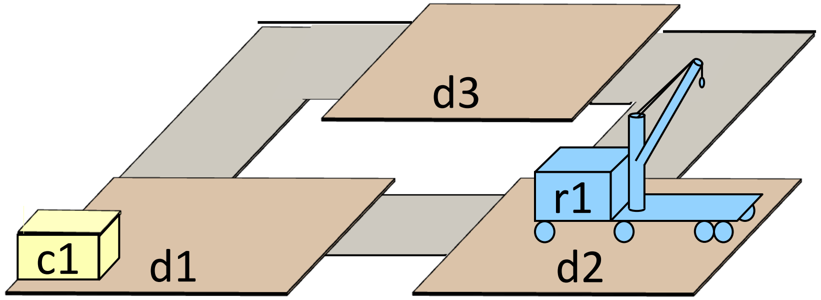
from $\hat{s}_0$:
- loc(c1) = d1
- loc(r1) = d1
- cargo(r1) = nil

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

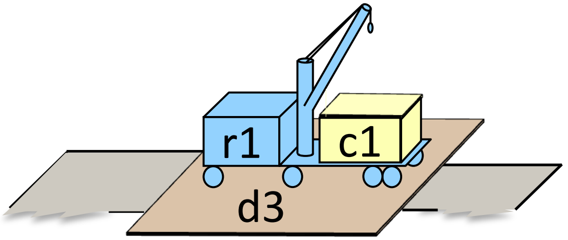
Example

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

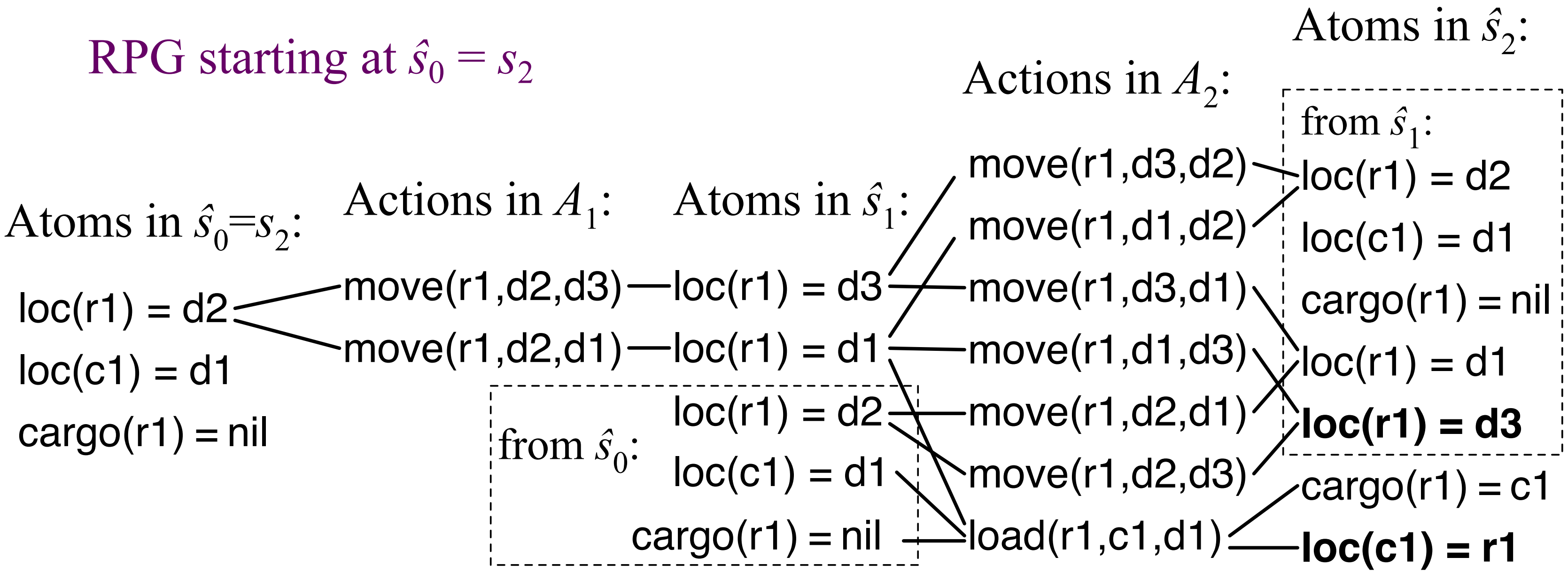
// *construct a relaxed solution* $\langle A_1, A_2, \ldots, A_k \rangle$:
$\hat{s}_0 \leftarrow s$
for $k = 1$ by 1 until $\hat{s}_k$ r-satisfies g
    $A_k \leftarrow$ {all actions r-applicable in $\hat{s}_{k-1}$}; $\hat{s}_k \leftarrow \gamma^{+}(s_{k-1}, A_k)$
    if $k > 1$ and $\hat{s}_k = \hat{s}_{k-1}$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



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



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

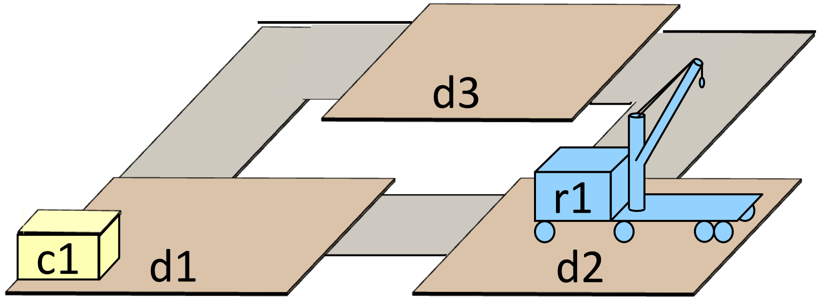


$\hat{s}_2$ r-satisfies g, so $\langle A_1, A_2 \rangle$ is a relaxed solution

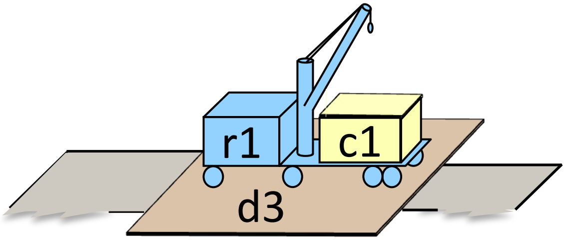
Example

- 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

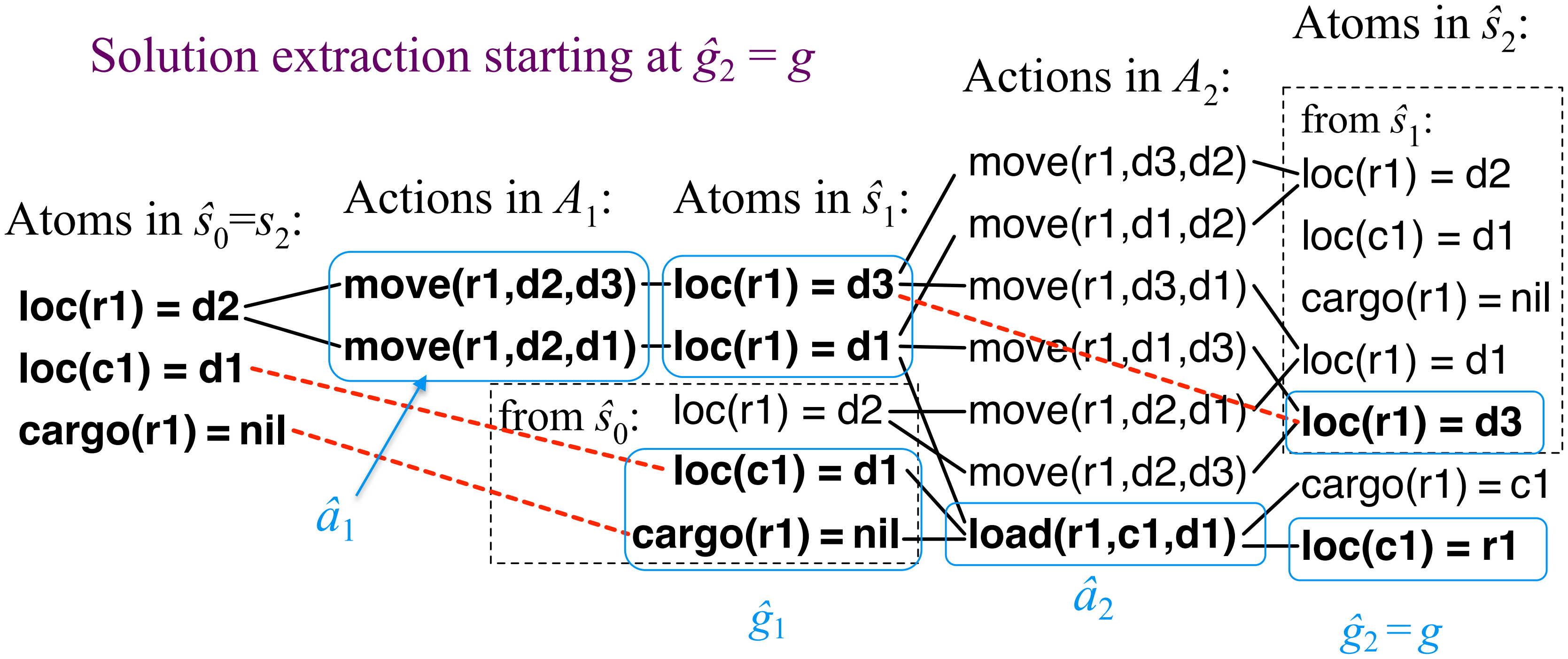
// *extract minimal relaxed solution* $\langle \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_k \rangle$:
$\hat{g}_k \leftarrow g$
for $i = k, k-1, \ldots, 1$:
  $\hat{a}_i \leftarrow$ any minimal subset of A_i such that $\gamma^+(\hat{s}_{i-1}, \hat{a}_i)$ r-satisfies $\hat{g}_i$
  $\hat{g}_{i-1} \leftarrow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



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



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



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

Properties

- Running time is polynomial in $|A| + \sum_{x \in X} |\text{Range}(x)|$

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

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

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

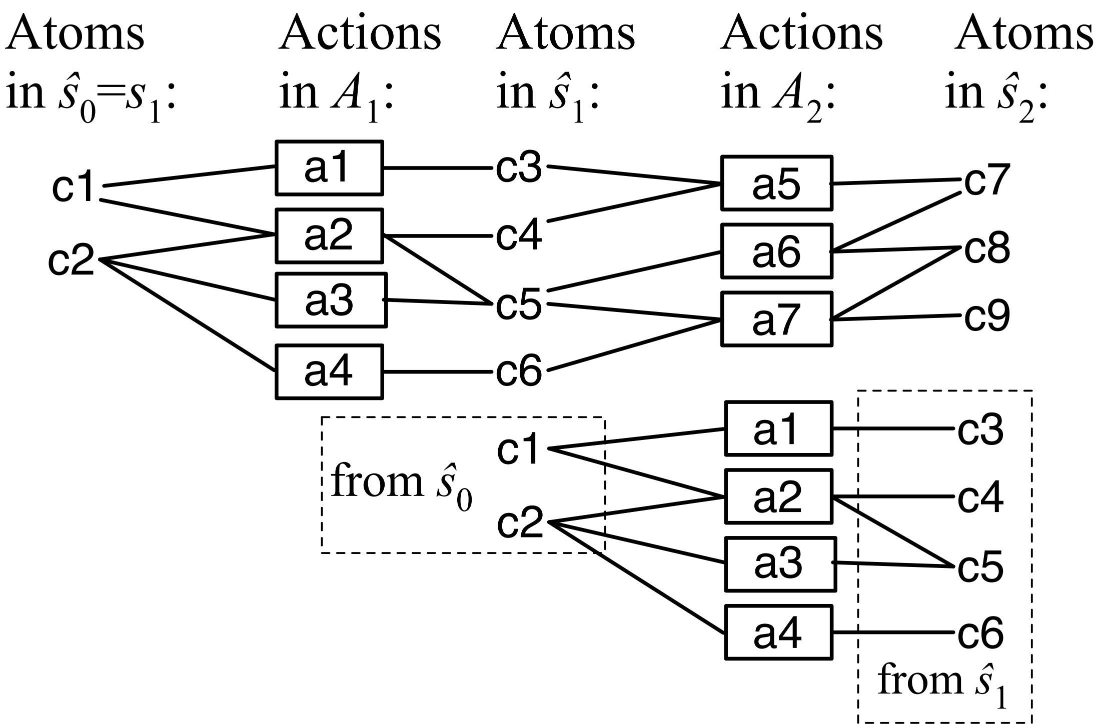
  - each $\hat{a}_i$ is a minimal set of actions such that $\gamma^+(\hat{s}_{i-1}, \hat{a}_i)$ r-satisfies $\hat{g}_i$
    - *minimal* doesn't mean *smallest*

- $h^{FF}(s)$ is ambiguous
  - depends on *which* minimal sets we choose

- h^{FF} not admissible

- $h^{FF}(s) \geq h^+(s)$ = *smallest* cost of any relaxed plan from s to goal

Example

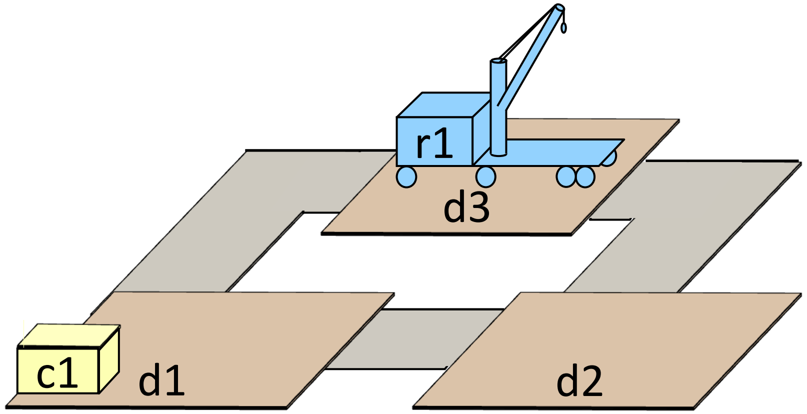


Poll. Suppose the goal atoms are c7, c8, c9. How many minimal relaxed solutions are there?

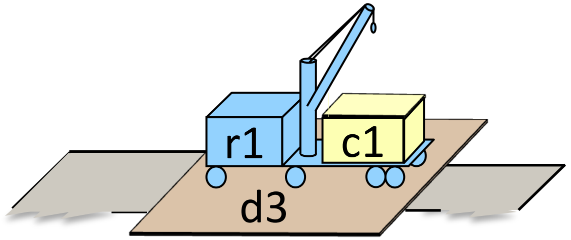
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. ≥ 8

3.2.2. 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}



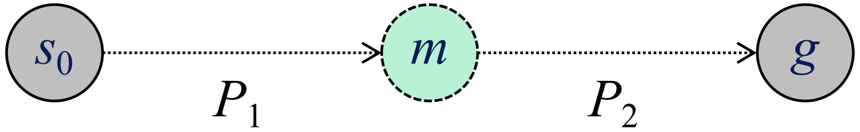
g = {loc(r1)=d3, loc(c1)=r1}

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

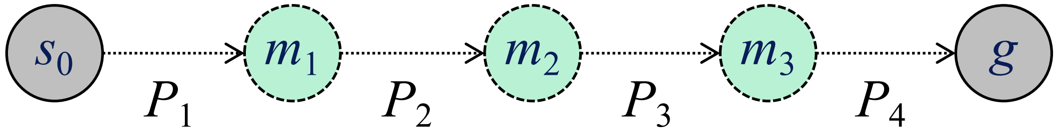
From now on, I'll abbreviate "disjunctive landmark" as "landmark"

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
  - …

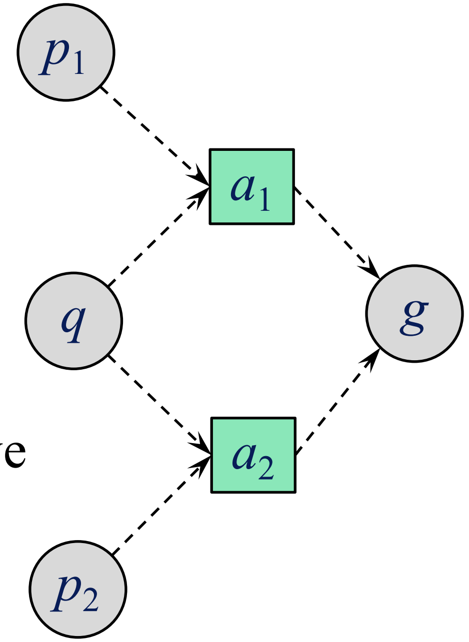
Computing Landmarks

- Given a formula φ
  - PSPACE-hard (worst case) to decide whether φ is a landmark
  - As hard as solving the planning problem itself

- 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} $= \{a_1, a_2\}$
    - $\text{pre}(a_1) = \{p_1, q\}$
    - $\text{pre}(a_2) = \{p_2, q\}$
  - To achieve g, must achieve $(p_1 \land q) \lor (p_2 \land q)$
    - same as $q \land (p_1 \lor p_2)$
  - Landmarks:
    - q
    - $p_1 \lor p_2$



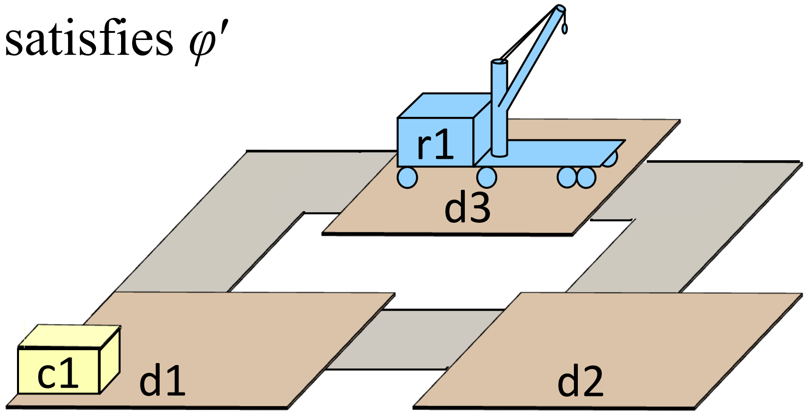
RPG-based Landmark Computation

- 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(Σ, s_0, g)

$Queue \leftarrow \langle$all literals in $g\rangle$; $Examined \leftarrow \emptyset$

while $Queue \neq \langle\rangle$ **do**

$\varphi \leftarrow$ pop($Queue$)

if $\varphi \notin Examined$ and $s_0 \not\models \varphi$ **then**

// *Step 1: look for an "action landmark"*

$R \leftarrow$ {actions whose effects include a literal in φ}

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

// *$\hat{s}_k$ now includes every atom that's achievable without R*

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

if $N = \emptyset$ then return failure

// *Step 2: get new landmarks from actions' preconditions*

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

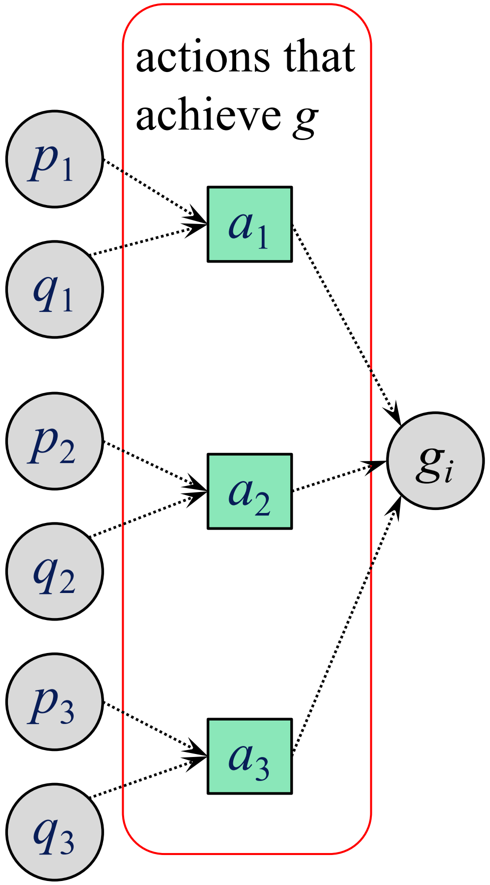
each p_i is a precondition of at least one $a \in N$, and

each $a \in N$ has at least one p_i as a precondition$\}$

append to $Queue$ every $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

add φ to $Examined$

return $Examined$



RPG-based Landmark Computation

RPG-Landmarks(Σ, s_0, g)

$Queue \leftarrow \langle$all literals in $g\rangle$; $Examined \leftarrow \emptyset$

while $Queue \neq \langle\rangle$ **do**

$\varphi \leftarrow$ pop($Queue$)

if $\varphi \notin Examined$ and $s_0 \not\models \varphi$ **then**

// Step 1: look for an "action landmark"

$R \leftarrow$ {actions whose effects include a literal in φ}

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

// $\hat{s}_k$ now includes every atom that's achievable without R

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

if $N = \emptyset$ then return failure

// Step 2: get new landmarks from actions' preconditions

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

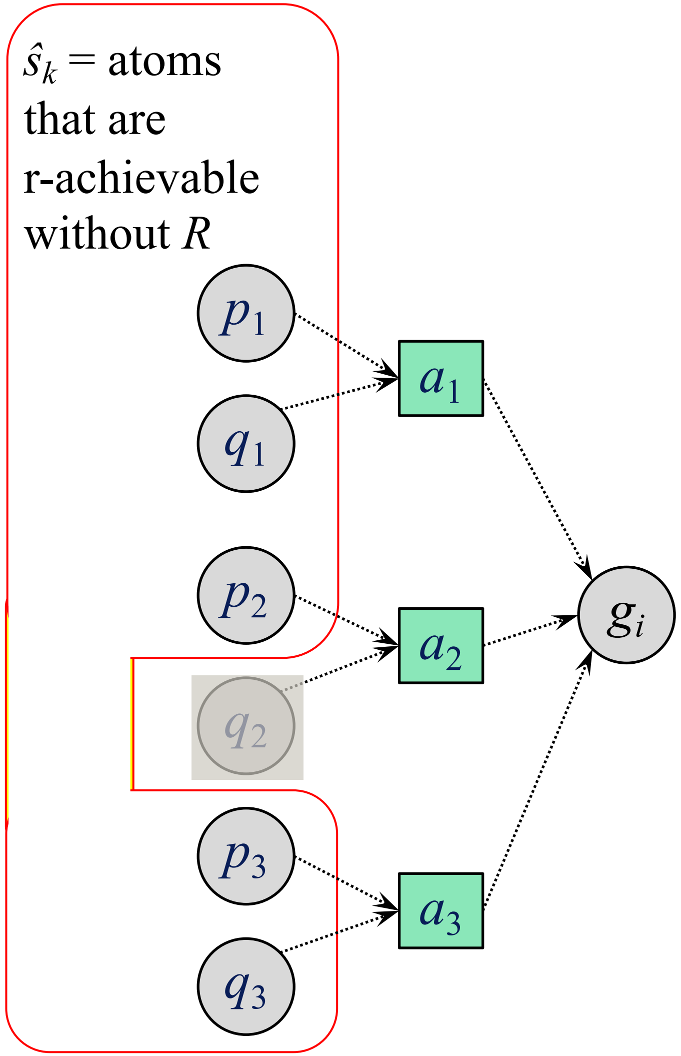
each p_i is a precondition of at least one $a \in N$, and

each $a \in N$ has at least one p_i as a precondition$\}$

append to $Queue$ every $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

add φ to $Examined$

return $Examined$



RPG-based Landmark Computation

RPG-Landmarks(Σ, s_0, g)

$Queue \leftarrow \langle$all literals in $g\rangle$; $Examined \leftarrow \emptyset$

while $Queue \neq \langle\rangle$ **do**

$\varphi \leftarrow$ pop($Queue$)

if $\varphi \notin Examined$ and $s_0 \not\models \varphi$ **then**

// Step 1: look for an "action landmark"

$R \leftarrow$ {actions whose effects include a literal in φ}

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

// $\hat{s}_k$ now includes every atom that's achievable without R

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

if $N = \emptyset$ then return failure

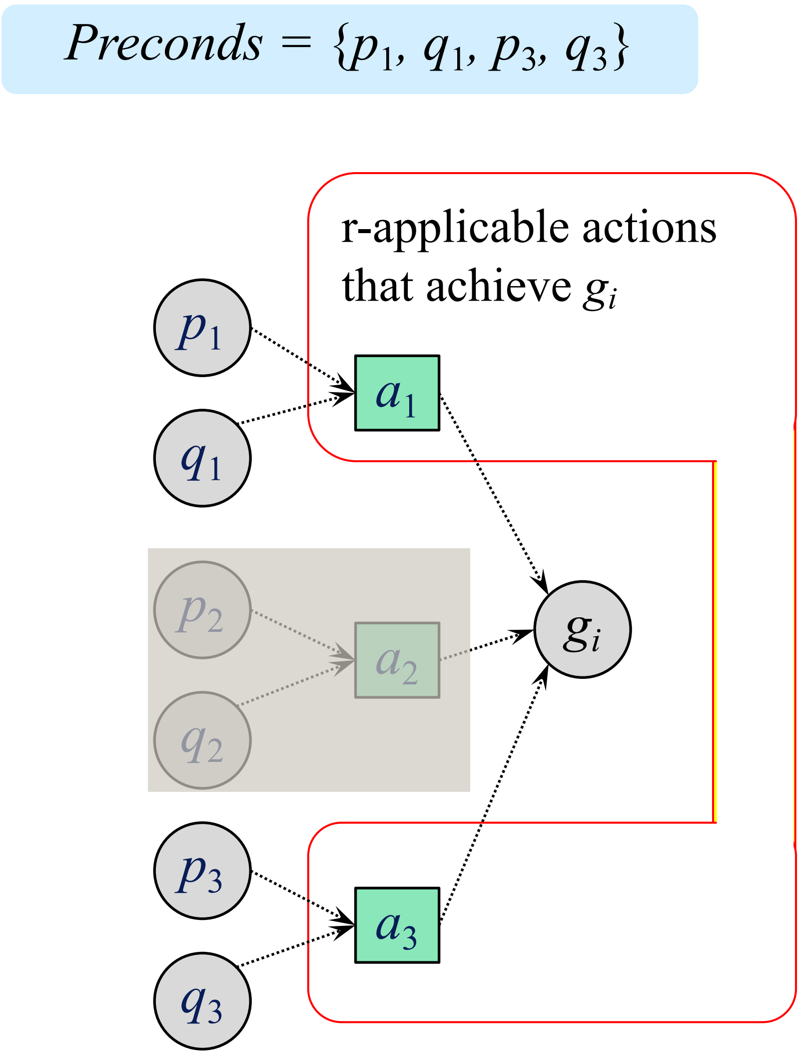
// Step 2: get new landmarks from actions' preconditions

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

append to $Queue$ every $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

add φ to $Examined$

return $Examined$



RPG-based Landmark Computation

RPG-Landmarks(Σ, s_0, g)

$Queue \leftarrow \langle$all literals in $g\rangle$; $Examined \leftarrow \emptyset$

while $Queue \neq \langle\rangle$ **do**

$\varphi \leftarrow$ pop($Queue$)

if $\varphi \notin Examined$ and $s_0 \not\models \varphi$ **then**

// Step 1: look for an "action landmark"

$R \leftarrow$ {actions whose effects include a literal in φ}

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

// $\hat{s}_k$ now includes every atom that's achievable without R

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

if $N = \emptyset$ then return failure

// Step 2: get new landmarks from actions' preconditions

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

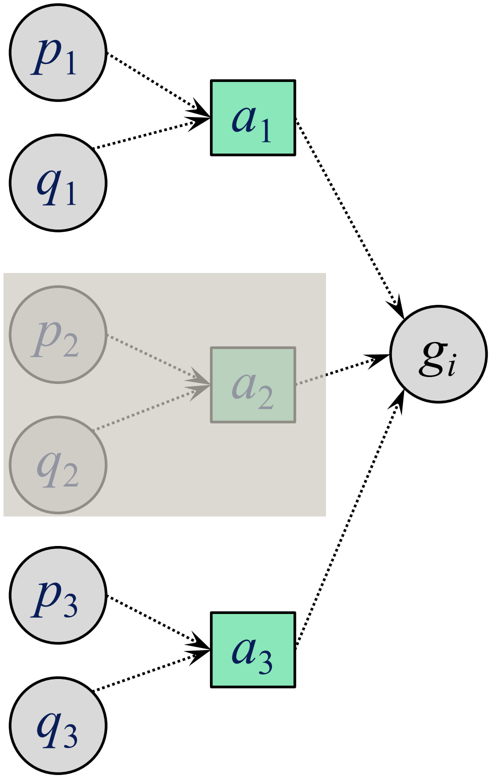
append to $Queue$ every $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

add φ to $Examined$

return $Examined$

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

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



RPG-Landmarks(Σ, s_0, g)

  $Queue \leftarrow \langle$all literals in $g\rangle$; $Examined \leftarrow \emptyset$
  while $Queue \neq \langle\rangle$ **do**
    $\varphi \leftarrow$ pop($Queue$)
    if $\varphi \notin Examined$ and $s_0 \not\models \varphi$ **then**

      // Step 1: look for an "action landmark"
      $R \leftarrow$ {actions whose effects include a literal in φ}
      generate RPG from s_0 using $A \setminus R$, stopping when $\hat{s}_k = \hat{s}_{k-1}$
      // $\hat{s}_k$ now includes every atom that's achievable without R
      $N \leftarrow$ {all actions in R that are r-applicable in $\hat{s}_k$}
      if $N = \emptyset$ then return failure

      // Step 2: get new landmarks from actions' preconditions
      $\Phi \leftarrow \{p_1 \vee p_2 \vee \ldots \vee p_m \mid m \leq 4$,
          each p_i is a precondition of at least one $a \in N$, and
          each $a \in N$ has at least one p_i as a precondition$\}$
      append to $Queue$ every $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$
      add φ to $Examined$
  return $Examined$

Example

$Queue$ = ⟨loc(r1)=d3, loc(c1)=r1⟩

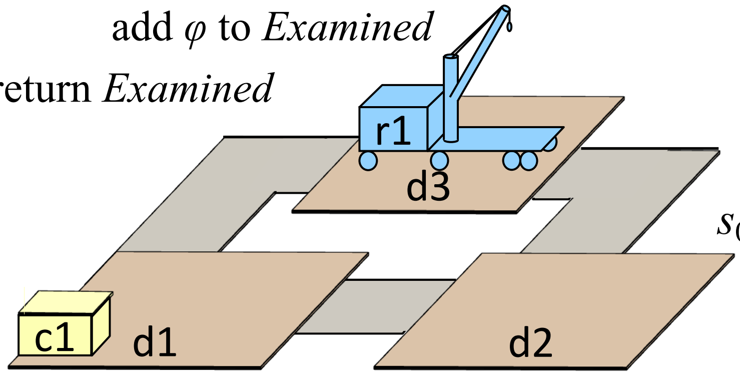
$Examined = \emptyset$

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

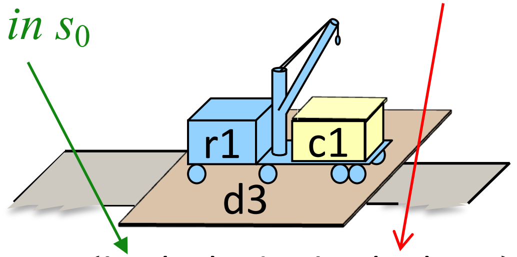
move(r, d, e)
  pre: loc(r)=d
  eff: loc(r)←e

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}

RPG-Landmarks(Σ, s_0, g)

$Queue \leftarrow \langle$all literals in $g\rangle$; $Examined \leftarrow \emptyset$

while $Queue \neq \langle\rangle$ **do**

$\varphi \leftarrow$ pop($Queue$)

if $\varphi \notin Examined$ and $s_0 \nvDash \varphi$ **then**

// Step 1: look for an "action landmark"

$R \leftarrow$ {actions whose effects include a literal in φ}

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

// $\hat{s}_k$ now includes every atom that's achievable without R

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

if $N = \emptyset$ then return failure

// Step 2: get new landmarks from actions' preconditions

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

each p_i is a precondition of at least one $a \in N$, and

each $a \in N$ has at least one p_i as a precondition}

append to $Queue$ every $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

add φ to $Examined$

return $Examined$

Example

$Queue$ = ⟨loc(c1)=r1⟩

$Examined = \emptyset$

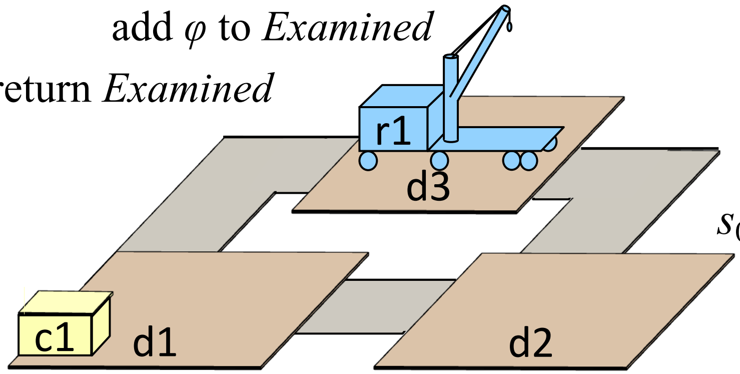
φ = loc(r1)=d3 ← $s_0 \vDash \varphi$

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

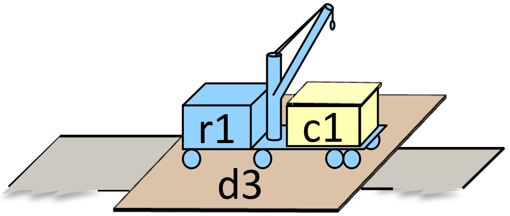
move(r, d, e)
pre: loc(r)=d
eff: loc(r)←e

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(Σ, s_0, g)

$Queue \leftarrow \langle$all literals in $g\rangle$; $Examined \leftarrow \emptyset$

while $Queue \neq \langle\rangle$ **do**

$\varphi \leftarrow$ pop($Queue$)

if $\varphi \notin Examined$ and $s_0 \nvDash \varphi$ **then**

// Step 1: look for an "action landmark"

$R \leftarrow$ {actions whose effects include a literal in φ}

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

// $\hat{s}_k$ now includes every atom that's achievable without R

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

if $N = \emptyset$ then return failure

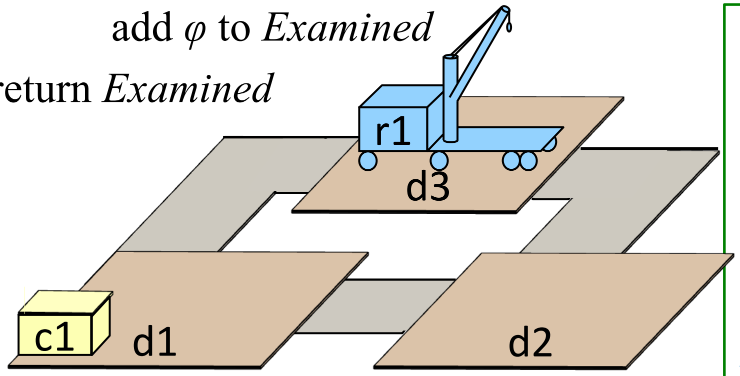
// Step 2: get new landmarks from actions' preconditions

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

append to $Queue$ every $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

add φ to $Examined$

return $Examined$



$Queue = \langle\rangle$

$Examined = \emptyset$

φ = loc(c1)=r1 ← $s_0 \nvDash \varphi$

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

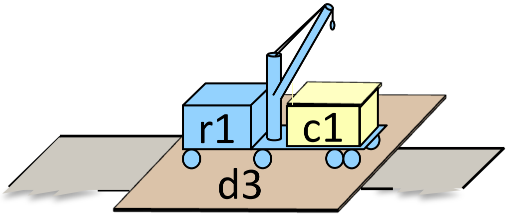
$r \in Robots$
$c \in Containers$
$l,d,e \in Locs$

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

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

both $\hat{s}_1$ and $\hat{s}_2$: loc(r1)=d1, loc(r1)=d2; 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(Σ, s_0, g)

$Queue \leftarrow \langle$all literals in $g\rangle$; $Examined \leftarrow \emptyset$

while $Queue \neq \langle\rangle$ **do**

$\varphi \leftarrow$ pop($Queue$)

if $\varphi \notin Examined$ and $s_0 \not\models \varphi$ **then**

// Step 1: look for an "action landmark"

$R \leftarrow$ {actions whose effects include a literal in φ}

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

// $\hat{s}_k$ now includes every atom that's achievable without R

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

if $N = \emptyset$ then return failure

// Step 2: get new landmarks from actions' preconditions

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

append to $Queue$ every $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

add φ to $Examined$

return $Examined$

$Queue = \langle\rangle$

$Examined = \emptyset$

φ = loc(c1)=r1

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

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

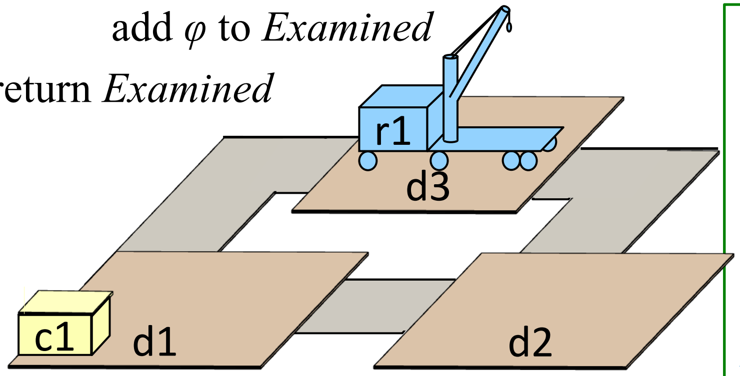
load(r, c, l)
pre: cargo(r)=nil, loc(c)=l, loc(r)=l
eff: cargo(r)←c, loc(c)←r

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

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

load (r1, c1, d)
pre: cargo(r1) = nil,
loc(c1) = d,
loc(r1) = d



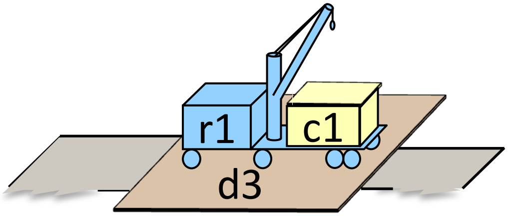
$\hat{s}_0$:
loc(c1)=d1
loc(r1)=d3
cargo(r1)=nil

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

both $\hat{s}_1$ and $\hat{s}_2$:
loc(r1)=d1
loc(r1)=d2

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}

RPG-Landmarks(Σ, s_0, g)

$Queue \leftarrow \langle$all literals in $g\rangle$; $Examined \leftarrow \emptyset$

while $Queue \neq \langle\rangle$ **do**

$\varphi \leftarrow$ pop($Queue$)

if $\varphi \notin Examined$ and $s_0 \not\models \varphi$ **then**

// Step 1: look for an "action landmark"

$R \leftarrow$ {actions whose effects include a literal in φ}

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

// $\hat{s}_k$ now includes every atom that's achievable without R

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

if $N = \emptyset$ then return failure

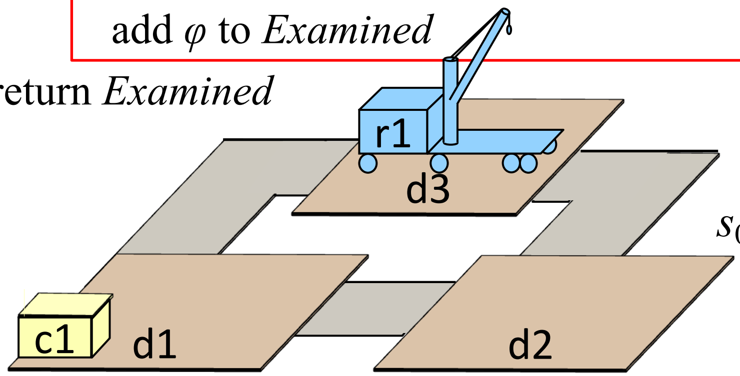
// Step 2: get new landmarks from actions' preconditions

$\Phi \leftarrow \{p_1 \lor p_2 \lor \ldots \lor p_m \mid m \leq 4$,
each p_i is a precondition of at least one $a \in N$, and
each $a \in N$ has at least one p_i as a precondition}

append to $Queue$ every $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

add φ to $Examined$

return $Examined$



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

Example

$Queue$ = ⟨cargo(r1)=nil, loc(c1)=d1, loc(r1)=d1⟩

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

φ = loc(c1)=r1

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

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

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

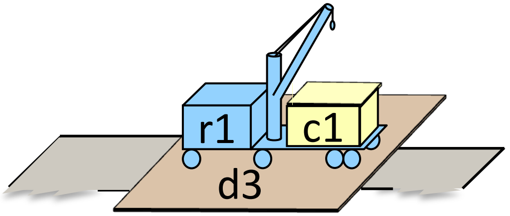
load (r1, c1, d1)
pre: cargo(r1) = nil,
loc(c1) = d1,
loc(r1) = d1

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}

RPG-Landmarks(Σ, s_0, g)

$Queue \leftarrow \langle$all literals in $g\rangle$; $Examined \leftarrow \emptyset$

while $Queue \neq \langle\rangle$ **do**

$\varphi \leftarrow$ pop($Queue$)

if $\varphi \notin Examined$ and $s_0 \nvDash \varphi$ **then**

// Step 1: look for an "action landmark"

$R \leftarrow$ {actions whose effects include a literal in φ}

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

// $\hat{s}_k$ now includes every atom that's achievable without R

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

if $N = \emptyset$ then return failure

// Step 2: get new landmarks from actions' preconditions

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

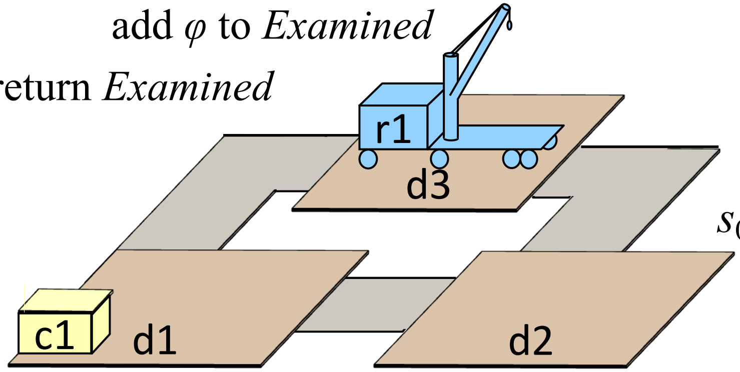
each p_i is a precondition of at least one $a \in N$, and

each $a \in N$ has at least one p_i as a precondition}

append to $Queue$ every $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

add φ to $Examined$

return $Examined$



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

Example

$Queue = \langle$ loc(c1)=d1, loc(r1)=d1$\rangle$

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

φ = cargo(r1)=nil ← $s_0 \vDash \varphi$

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

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

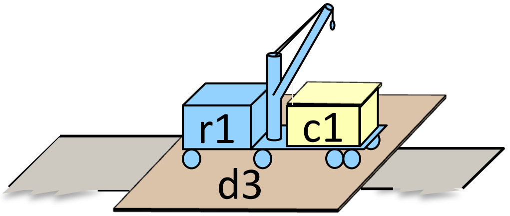
Φ = {cargo(r1)=nil, loc(c1)=d1, loc(r1)=d1, …}

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}

RPG-Landmarks(Σ, s_0, g)

Queue ← ⟨all literals in g⟩; *Examined* ← ∅

while *Queue* ≠ ⟨⟩ **do**

φ ← pop(*Queue*)

if $\varphi \notin$ *Examined* and $s_0 \nvDash \varphi$ **then**

// Step 1: look for an "action landmark"

R ← {actions whose effects include a literal in φ}

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

// $\hat{s}_k$ now includes every atom that's achievable without R

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

if $N = \emptyset$ then return failure

// Step 2: get new landmarks from actions' preconditions

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

each p_i is a precondition of at least one $a \in N$, and

each $a \in N$ has at least one p_i as a precondition}

append to *Queue every* $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

add φ to *Examined*

return *Examined*

Example

Queue = ⟨ loc(r1)=d1⟩

Examined = {loc(c1)=r1}

φ = loc(c1)=d1 ← $s_0 \vDash \varphi$

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

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

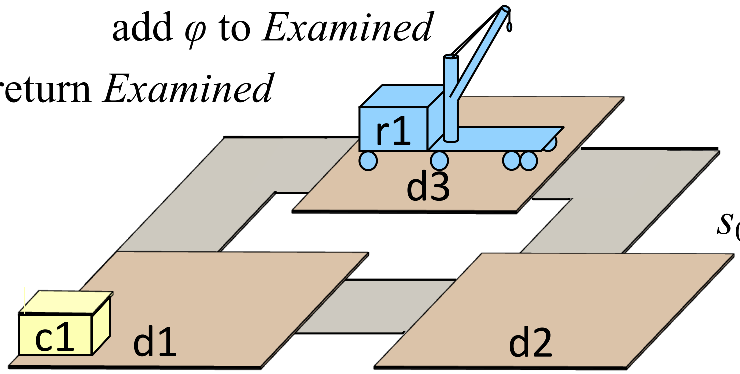
Φ = {cargo(r1)=nil, loc(c1)=d1, loc(r1)=d1, …}

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

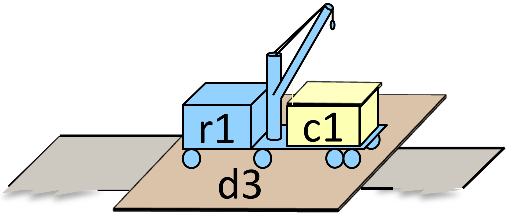
move(r, d, e)
  pre: loc(r)=d
  eff: loc(r)←e

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*



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



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

Example

RPG-Landmarks(Σ, s_0, g)

$Queue \leftarrow$ ⟨all literals in g⟩; $Examined \leftarrow \emptyset$

while $Queue \neq \langle\rangle$ **do**

$\varphi \leftarrow$ pop($Queue$)

if $\varphi \notin Examined$ and $s_0 \not\models \varphi$ **then**

// Step 1: look for an "action landmark"

$R \leftarrow$ {actions whose effects include a literal in φ}

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

// $\hat{s}_k$ now includes every atom that's achievable without R

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

if $N = \emptyset$ then return failure

// Step 2: get new landmarks from actions' preconditions

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

each p_i is a precondition of at least one $a \in N$, and

each $a \in N$ has at least one p_i as a precondition}

append to $Queue$ every $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

add φ to $Examined$

return $Examined$

$Queue = \langle\rangle$

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

φ = loc(r1)=d1 ← $s_0 \not\models \varphi$

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

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

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

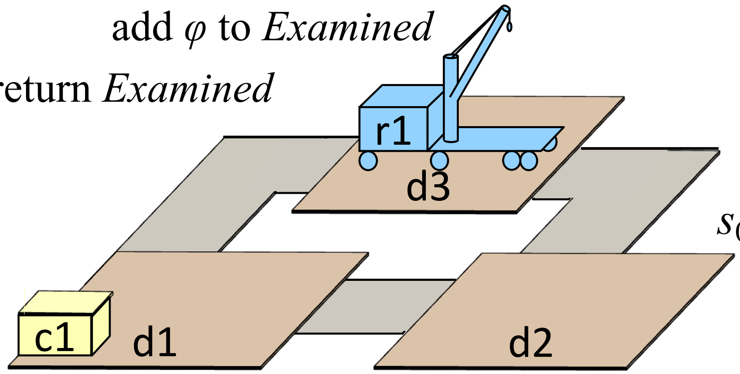
$A \setminus R$ = {load(r1,c1,l), unload(r1,c1,l), move(r1,d,d2), move(r1,d,d3)}

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

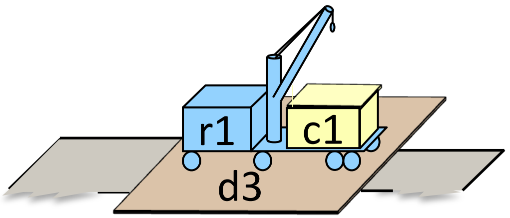
unload(r, c, l)
pre: loc(c)=r, loc(r)=l
eff: cargo(r)←nil, loc(c)←l

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



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

$\hat{s}_k$ = {loc(r1)=d2, loc(r1)=d3, loc(c1)=d1, cargo(r1)=nil}



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

Example

RPG-Landmarks(Σ, s_0, g)

  $Queue \leftarrow \langle$all literals in $g\rangle$; $Examined \leftarrow \emptyset$

  while $Queue \neq \langle\rangle$ **do**

    $\varphi \leftarrow$ pop($Queue$)

    if $\varphi \notin Examined$ and $s_0 \not\models \varphi$ **then**

      // Step 1: look for an "action landmark"

      $R \leftarrow$ {actions whose effects include a literal in φ}

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

      // $\hat{s}_k$ now includes every atom that's achievable without R

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

      if $N = \emptyset$ then return failure

      // Step 2: get new landmarks from actions' preconditions

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

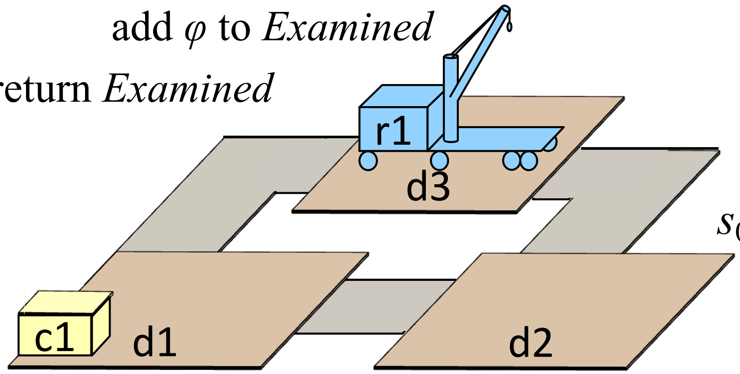
        each p_i is a precondition of at least one $a \in N$, and

        each $a \in N$ has at least one p_i as a precondition}

      append to $Queue$ every $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

      add φ to $Examined$

  return $Examined$



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

$Queue = \langle\rangle$

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

φ = loc(r1)=d1

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

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

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

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

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

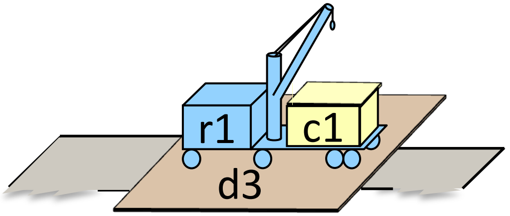
$\hat{s}_k$ = {loc(r1)=d2, loc(r1)=d3,
loc(c1)=d1,
cargo(r1)=nil}

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(Σ, s_0, g)

    Queue ← ⟨all literals in g⟩; *Examined* ← ∅

    while *Queue* ≠ ⟨⟩ **do**

        φ ← pop(*Queue*)

        if $\varphi \notin$ *Examined* and $s_0 \not\models \varphi$ **then**

            // Step 1: look for an "action landmark"

            R ← {actions whose effects include a literal in φ}

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

            // $\hat{s}_k$ now includes every atom that's achievable without R

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

            if $N = \emptyset$ then return failure

            // Step 2: get new landmarks from actions' preconditions

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

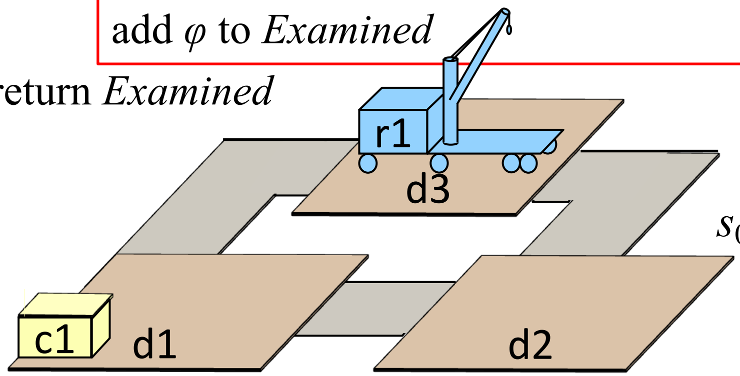
                each p_i is a precondition of at least one $a \in N$, and

                each $a \in N$ has at least one p_i as a precondition}

            append to *Queue every* $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

            add φ to *Examined*

    return *Examined*



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

Queue = ⟨loc(r1)=d2 ∨ loc(r1)=d3⟩

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

φ = loc(r1)=d1

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

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

Φ = {loc(r1)=d2 ∨ loc(r1)=d3}

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

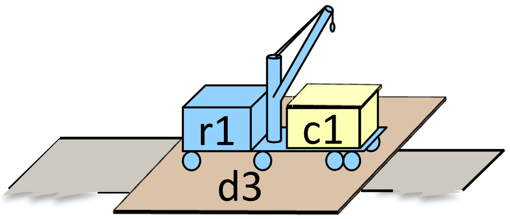
move(r1, d3, d1)
    pre: loc(r1) = d3
    eff: loc(r1) ← d1

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(Σ, s_0, g)

$Queue \leftarrow \langle$all literals in $g\rangle$; $Examined \leftarrow \emptyset$

while $Queue \neq \langle\rangle$ **do**

$\varphi \leftarrow$ pop($Queue$)

if $\varphi \notin Examined$ and $s_0 \nvDash \varphi$ **then**

// Step 1: look for an "action landmark"

$R \leftarrow$ {actions whose effects include a literal in φ}

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

// $\hat{s}_k$ now includes every atom that's achievable without R

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

if $N = \emptyset$ then return failure

// Step 2: get new landmarks from actions' preconditions

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

each p_i is a precondition of at least one $a \in N$, and

each $a \in N$ has at least one p_i as a precondition}

append to $Queue$ every $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

add φ to $Examined$

return $Examined$

$Queue = \langle\rangle$

$Examined = \{$loc(c1)=r1, loc(r1)=d1$\}$

φ = loc(r1)=d2 ∨ loc(r1)=d3

$R = \{$move(r1,d2,d1), move(r1,d3,d1)$\}$ ↑ $s_0 \vDash \varphi$

$N = \{$move(r1,d2,d1) move(r1,d3,d1)$\}$

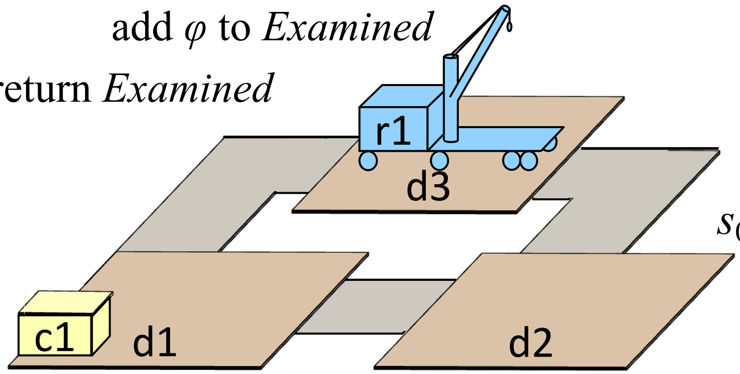
$\Phi = \{$loc(r1)=d2 ∨ loc(r1)=d3$\}$

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

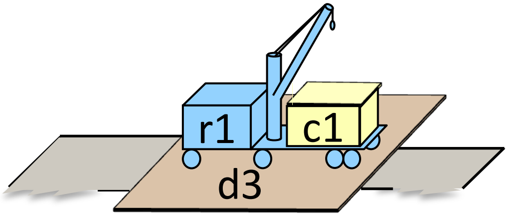
move(r, d, e)
pre: loc(r)=d
eff: loc(r)←e

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

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



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



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

Example

RPG-Landmarks(Σ, s_0, g)

$\quad Queue \leftarrow \langle$all literals in $g\rangle$; $Examined \leftarrow \emptyset$

$\quad$ **while** $Queue \neq \langle\rangle$ **do**

$\quad\quad \varphi \leftarrow$ pop($Queue$)

$\quad\quad$ **if** $\varphi \notin Examined$ and $s_0 \not\models \varphi$ **then**

$\quad\quad\quad$ *// Step 1: look for an "action landmark"*

$\quad\quad\quad R \leftarrow$ {actions whose effects include a literal in φ}

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

$\quad\quad\quad$ *//* $\hat{s}_k$ *now includes every atom that's achievable without* R

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

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

$\quad\quad\quad$ *// Step 2: get new landmarks from actions' preconditions*

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

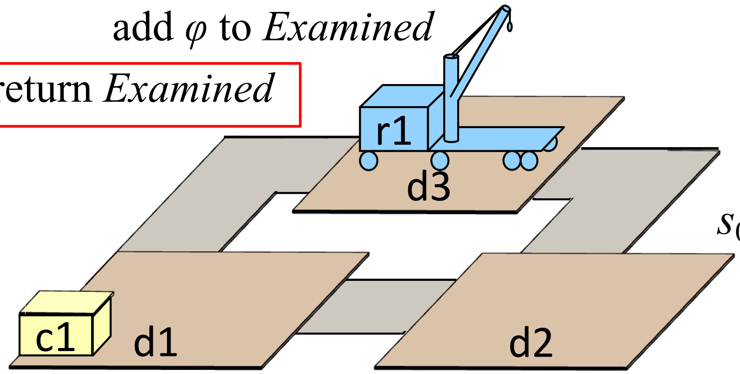
$\quad\quad\quad\quad$ each p_i is a precondition of at least one $a \in N$, and

$\quad\quad\quad\quad$ each $a \in N$ has at least one p_i as a precondition}

$\quad\quad\quad$ append to $Queue$ every $\varphi \in \Phi$ that isn't subsumed by another $\varphi' \in \Phi$

$\quad\quad\quad$ add φ to $Examined$

$\quad$ return $Examined$



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

$Queue = \langle\rangle$

Return 2

$Examined$ = {loc(c1)=r1, loc(r1)=d1}

φ = loc(r1)=d2 ∨ loc(r1)=d3

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

N = {move(r1,d2,d1)}

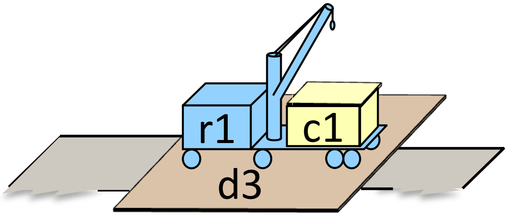
Φ = {loc(r1)=d2}

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}

Summary

- Search-tree terminology
- 3.1. Forward Search
  - Forward-search, Forward-Search-Det
  - cycle-checking
  - Search algorithms classified by
    - (i) node selection
    - (ii) pruning
  - Breadth-first, depth-first, uniform-cost search
  - A*, GBFS
  - DFBB, IDS
- 3.2. Heuristic Functions
  - Straight-line distance example
  - Delete-relaxation heuristics
    - relaxed states, γ^+,
    - h^+ – minimal relaxed solution heuristic
    - h^{FF} – Fast-Forward heuristic
    - HFF algorithm – computes h^{FF}
  - Disjunctive landmarks, RPG-Landmark, h^{RL}
    - Get necessary actions by making RPG for all non-relevant actions