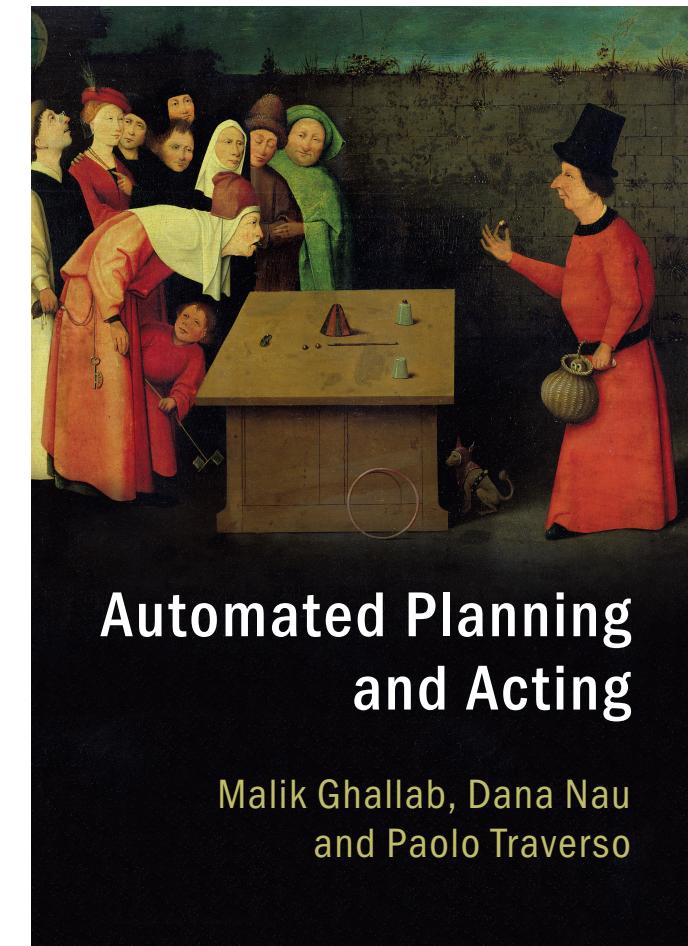


# EWS kau temă 4

## Chapter 2 Deliberation with Deterministic Models

Sections 2.1, 2.2, 2.6

Dana Nau  
University of Maryland



# Motivation

- How to model a complex environment?
  - ▶ Generally need simplifying assumptions
- *Classical planning*
  - Finite, static world, just one actor
  - No concurrent actions, no explicit time
  - Determinism, no uncertainty
  - ▶ Sequence of states and actions  $\langle s_0, a_1, s_1, a_2, s_2, \dots \rangle$
- Avoids many complications
- Most real-world environments don't satisfy the assumptions
  - ⇒ Errors in prediction
- OK if they're infrequent and don't have severe consequences

# Outline

- Next →*
- Chapter 2, part *a* (chap2a.pdf):
    - 2.1 State-variable representation
    - Comparison with PDDL
    - 2.2 Forward state-space search
    - 2.6 Incorporating planning into an actor
  - Chapter 2, part *b* (chap2b.pdf):
    - 2.3 Heuristic functions
    - 2.7.7 HTN planning
  - Chapter 2, part *c* (chap2c.pdf):
    - 2.4 Backward search
    - 2.5 Plan-space search
  - Additional slides:
    - 2.7.8 LTL\_planning.pdf

# Domain Model

*State-transition system or classical planning domain:*

- $\Sigma = (S, A, \gamma, \text{cost})$  or  $(S, A, \gamma)$

- ▶  $S$  - finite set of *states*
- ▶  $A$  - finite set of *actions*
- ▶  $\gamma: S \times A \rightarrow S$

*prediction (or state-transition) function*

- *partial* function:  $\gamma(s, a)$  is not necessarily defined for every  $(s, a)$ 
  - ▶  $a$  is *applicable* in  $s$  iff  $\gamma(s, a)$  is defined
  - ▶  $\text{Domain}(a) = \{s \in S \mid a \text{ is applicable in } s\}$
  - ▶  $\text{Range}(a) = \{\gamma(s, a) \mid s \in \text{Domain}(a)\}$
- ▶  $\text{cost}: S \times A \rightarrow \mathbb{R}^+$  or  $\text{cost}: A \rightarrow \mathbb{R}^+$ 
  - optional; default is  $\text{cost}(a) \equiv 1$
  - money, time, something else

- *plan*:

- ▶ a sequence of actions  $\pi = \langle a_1, \dots, a_n \rangle$

- $\pi$  is *applicable* in  $s_0$  if the actions are applicable in the order given

$$\gamma(s_0, a_1) = s_1$$

$$\gamma(s_1, a_2) = s_2$$

...

$$\gamma(s_{n-1}, a_n) = s_n$$

- ▶ In this case define  $\gamma(s_0, \pi) = s_n$

- *Classical planning problem*:

- ▶  $P = (\Sigma, s_0, S_g)$

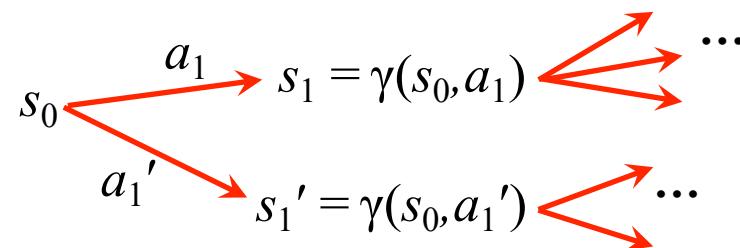
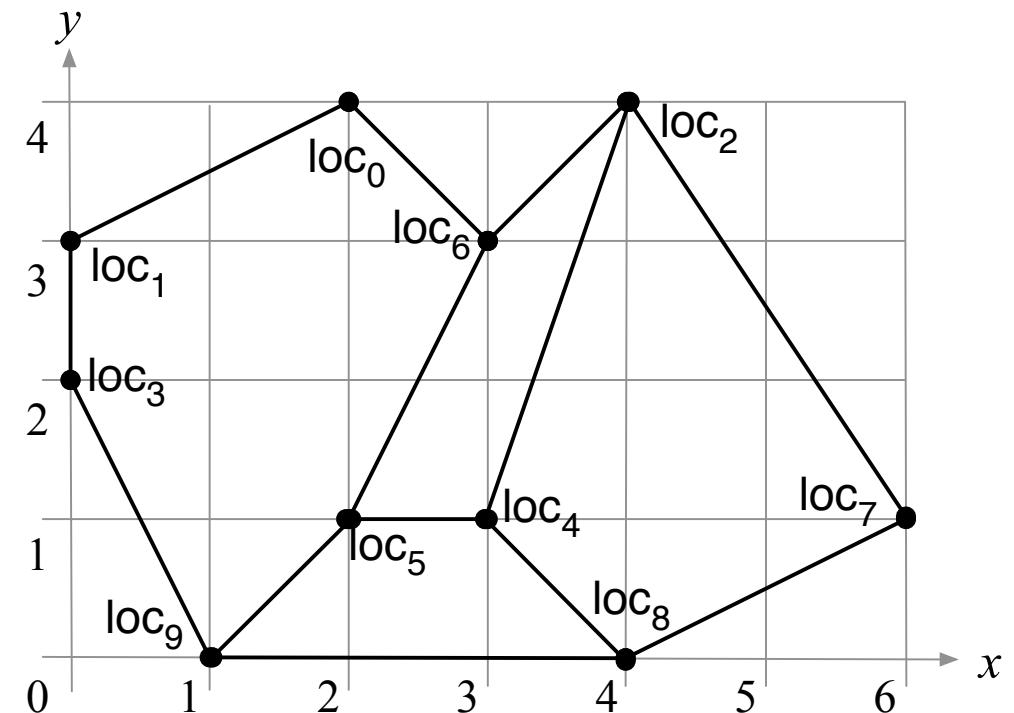
- ▶ planning domain, initial state, set of goal states

- *Solution* for  $P$ :

- ▶ a plan  $\pi$  such that  $\gamma(s_0, \pi) \in S_g$

# Representing $\Sigma$

- If  $S$  and  $A$  are small enough
  - ▶ Give each state and action a name
  - ▶ For each  $s$  and  $a$ , store  $\gamma(s,a)$  in a lookup table
- In larger domains, don't represent all states explicitly
  - ▶ Language for describing properties of states
  - ▶ Language for describing how each action changes those properties
  - ▶ Start with initial state, use actions to produce other states

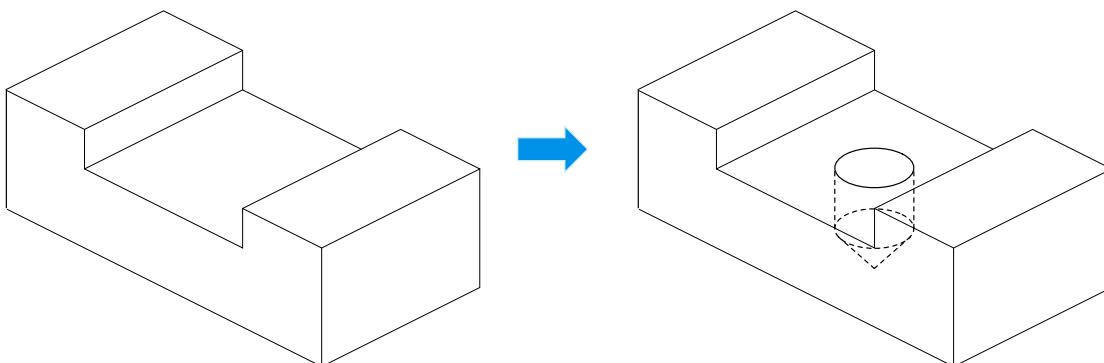


# Domain-Specific Representation

- Tailor-made for a specific environment
- State: arbitrary data structure *lists, tables etc.*
- Action: (head, preconditions, effects, cost)
  - ▶ *head*: name and parameter list
    - Get actions by instantiating the parameters
  - ▶ *preconditions*:
    - Computational tests to predict whether an action can be performed
    - Should be necessary/sufficient for the action to run without error
  - ▶ *effects*:
    - Procedures that modify the current state
  - ▶ *cost*: procedure that returns a number
    - Can be omitted, default is cost  $\equiv 1$

# Example

- Drilling holes in a metal workpiece
  - ▶ A state
    - geometric model of the workpiece
      - ▶ *annotated* with dimensions, tolerances, etc.
    - capabilities and status of drilling machine and drill bit
  - ▶ Several actions
    - clamp the workpiece onto the drilling machine
    - load a drill bit into the machine
    - drill a hole
- Name: drill-hole
- Arguments:
  - ▶ ID codes for the machine and drill bit
  - ▶ annotated geometric model of the workpiece
  - ▶ description of the hole to be drilled
- Preconditions
  - ▶ *Capabilities*: can the machine and drill bit produce the desired hole?
  - ▶ *Current state*: Is the drill bit installed? Is the workpiece clamped onto the table? Etc.
- Effects
  - ▶ annotated geometric model of modified workpiece
- Cost
  - ▶ estimate of time or monetary cost

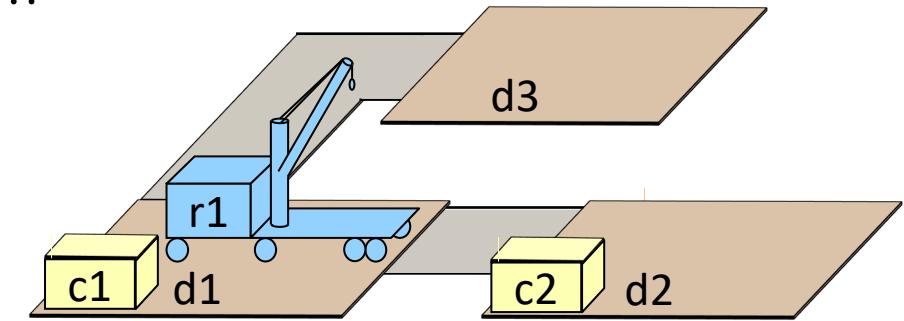


# Discussion

- Advantage of domain-specific representation:
  - ▶ use whatever works best for that particular domain
- Disadvantage:
  - ▶ for each new domain, need new representation and deliberation algorithms
- Alternative: *domain-independent* representation
  - ▶ Try to create a “standard format” that can be used for many different planning domains
  - ▶ Deliberation algorithms that work for anything in this format
- *State-variable* representation
  - ▶ Simple formats for describing states and actions
  - ▶ Limited representational capability
    - But easy to compute, easy to reason about
  - ▶ Domain-independent search algorithms and heuristic functions that can be used in all state-variable planning problems

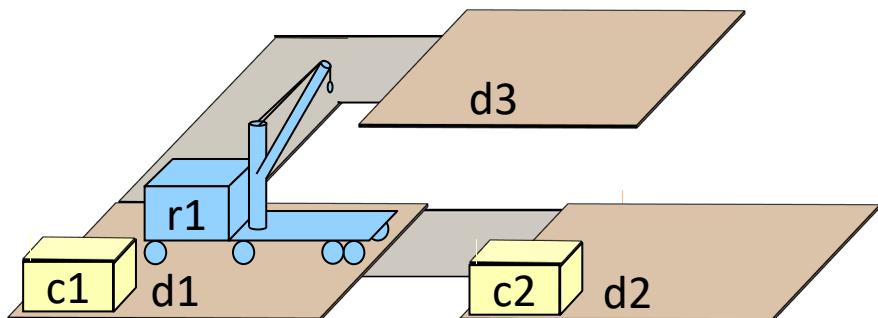
# State-Variable Representation

- $E$ : environment that we want to represent
- $B$ : set of symbols called *objects*
  - ▶ names for objects in  $E$ , mathematical constants, ...
- Example
  - ▶  $B = \text{Robots} \cup \text{Containers} \cup \text{Locs} \cup \{\text{nil}\}$ 
    - $\text{Robots} = \{r1\}$
    - $\text{Containers} = \{c1, c2\}$
    - $\text{Locs} = \{d1, d2, d3\}$
- $B$  only needs to include objects that matter at the current level of abstraction
- Can omit lots of details
  - ▶ physical characteristics of robots, containers, loading docks, roads, ...



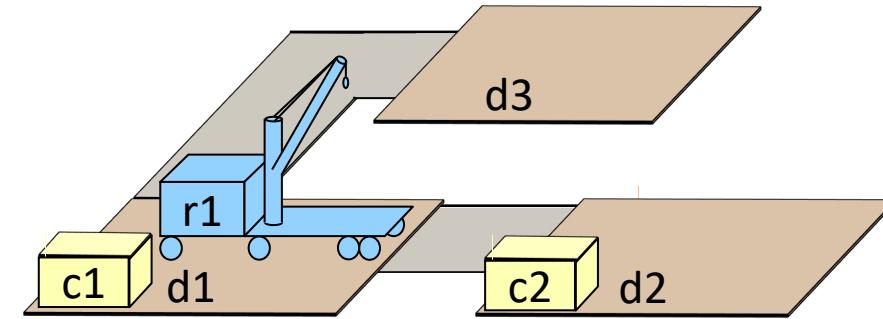
# Rigid Properties

- Objects have two kinds of properties
  - ▶ *rigid* and *varying*
- *Rigid*: stays the same in every state
  - ▶ Can be described as a mathematical relation  
 $\text{adjacent} = \{(d1,d2), (d2,d1), (d1,d3), (d3,d1)\}$
  - ▶ Or equivalently, a set of *ground atoms*  
 $\text{adjacent}(d1,d2), \text{adjacent}(d2,d1),$   
 $\text{adjacent}(d1,d3), \text{adjacent}(d3,d1)$
  - ▶ I'll use the two notations interchangeably
- Terminology from first-order logic:
  - ▶ *atom*  $\equiv$  *atomic formula*  $\equiv$  *positive literal*  
 $\equiv$  predicate symbol with list of arguments
    - e.g.,  $\text{adjacent}(x,d2)$
  - ▶ an atom is *ground* (or *fully instantiated*) if it contains no variable symbols
    - e.g.,  $\text{adjacent}(d1,d2)$
  - ▶ *negative literal*  $\equiv$  *negated atom*  $\equiv$  atom with a negation sign in front of it
    - e.g.,  $\neg \text{adjacent}(x,d2)$



# Varying Properties

- *Varying* property (or *fluent*):
  - a property that may differ in different states
- Represent it using a *state variable*
  - ▶ a term that we can assign a value to
    - e.g.,  $\text{loc}(r1)$
- Let  $X = \{\text{all state variables in the environment}\}$ 
  - e.g.,  $X = \{\text{loc}(r1), \text{loc}(c1), \text{loc}(c2), \text{cargo}(r1)\}$
- Each state variable  $x \in X$  has a *range*
  - = {all values that can be assigned to  $x$ }
  - Range( $\text{loc}(r1)$ ) = *Locs*
  - Range( $\text{loc}(c1)$ ) = Range( $\text{loc}(c2)$ ) = *Robots*  $\cup$  *Locs*
  - Range( $\text{cargo}(r1)$ ) = *Containers*  $\cup$  {nil}
- To abbreviate the “range” notation often I’ll just say things like
  - ▶  $\text{loc}(r1) \in \text{Locs}$
  - ▶  $\text{loc}(c1), \text{loc}(c2) \in \text{Robots} \cup \text{Locs}$



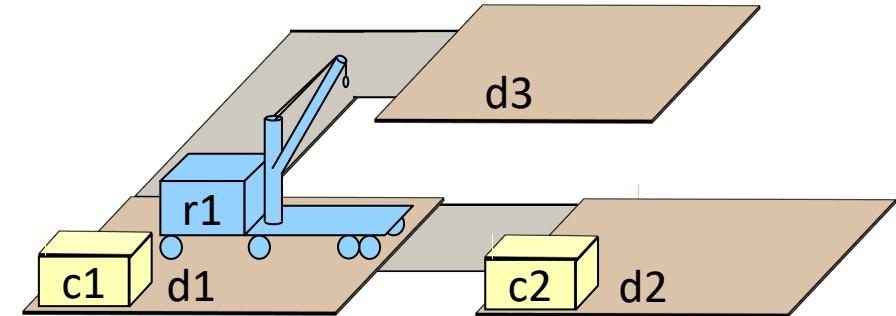
Instead of “domain”,  
to avoid confusion  
with planning domains

# States as Functions

- Represent each state  $s$  as a function that assigns values to state variables
  - ▶ For each state variable  $x$ ,  $s(x)$  is one  $x$ 's possible values

$$s_1(\text{loc}(r1)) = d1, \quad s_1(\text{cargo}(r1)) = \text{nil},$$

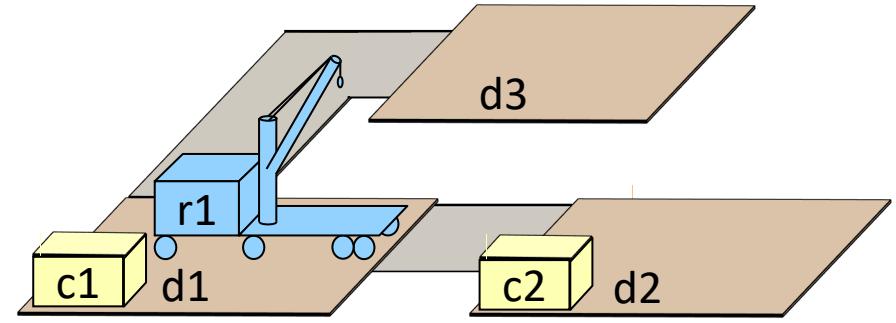
$$s_1(\text{loc}(c1)) = d1, \quad s_1(\text{loc}(c2)) = d2$$



- Mathematically, a function is a set of ordered pairs
$$s_1 = \{(\text{loc}(r1), d1), (\text{cargo}(r1), \text{nil}), (\text{loc}(c1), d1), (\text{loc}(c2), d2)\}$$
- Equivalently, write it as a set of *ground positive literals* (or *ground atoms*):
$$s_1 = \{\text{loc}(r1)=d1, \text{cargo}(r1)=\text{nil}, \text{loc}(c1)=d1, \text{loc}(c2)=d2\}$$
  - ▶ Here, we're using '=' as a predicate symbol

# Action Templates

- Action *template* or *schema*: a parameterized set of actions  
 $\alpha = (\text{head}, \text{pre}, \text{eff}, \text{cost})$ 
  - ▶ head: *name, parameters*
  - ▶ pre: *precondition literals*
  - ▶ eff: *effect literals*
  - ▶ cost: *a number* (optional, default is 1)
- e.g.,
  - ▶ head = take( $r;l,c$ )
  - ▶ pre = {cargo( $r$ )=nil, loc( $r$ )= $l$ , loc( $c$ )= $l$ }
  - ▶ eff = {cargo( $r$ )= $c$ , loc( $c$ )= $r$ }
- Each parameter has a range of possible values.
  - ▶ Range( $r$ ) = *Robots* = {r1}
  - ▶ Range( $l$ ) = *Locs* = {d1,d2,d3}
  - ▶ Range( $l$ ) = Range( $m$ ) = *Locs* = {d1,d2,d3}
  - ▶ Range( $c$ ) = *Containers* = {c1,c2}
- But we'll usually write it more like pseudocode



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

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

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

 $r \in \text{Robots} = \{\text{r1}\}$ 
 $l,m \in \text{Locs} = \{\text{d1},\text{d2},\text{d3}\}$ 
 $c \in \text{Containers} = \{\text{c1},\text{c2}\}$ 
```

# Actions

- $\mathcal{A}$  = set of action templates

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

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

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

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

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

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

$r \in \text{Robots} = \{\text{r1}\}$

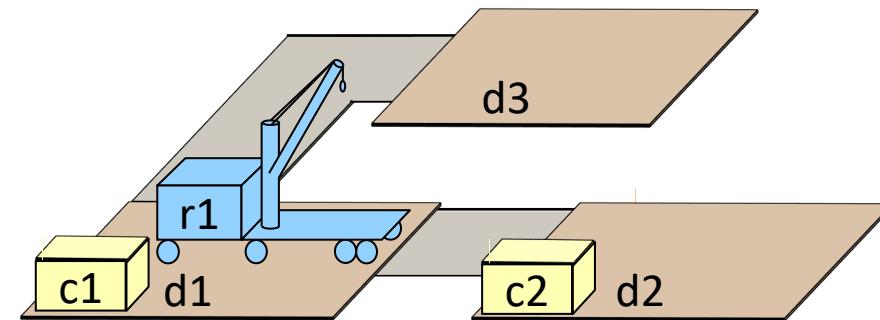
$l,m \in \text{Locs} = \{\text{d1}, \text{d2}, \text{d3}\}$

$c \in \text{Containers} = \{\text{c1}, \text{c2}\}$

- Action: *ground instance* of an  $\alpha \in \mathcal{A}$ 
  - ▶ replace each parameter with something in its range
- $A = \{\text{all actions we can get from } \mathcal{A}\}$   
=  $\{\text{all ground instances of members of } \mathcal{A}\}$

$\text{move(r1,d1,d2)}$

pre:  $\text{loc(r1)}=\text{d1}$ ,  $\text{adjacent(d1,d2)}$   
eff:  $\text{loc(r1)} \leftarrow \text{d2}$



# Actions

- $\mathcal{A}$  = set of action templates

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

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

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

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

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

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

$r \in \text{Robots} = \{\text{r1}\}$

$l,m \in \text{Locs} = \{\text{d1,d2,d3}\}$

$c \in \text{Containers} = \{\text{c1,c2}\}$

- Action: *ground instance* of an  $\alpha \in \mathcal{A}$ 
  - ▶ replace each parameter with something in its range
- $A = \{\text{all actions we can get from } \mathcal{A}\}$   
 $= \{\text{all ground instances of members of } \mathcal{A}\}$

$\text{move(r1,d1,d2)}$

pre:  $\text{loc(r1)}=\text{d1}$ ,  $\text{adjacent(d1,d2)}$   
eff:  $\text{loc(r1)} \leftarrow \text{d2}$

**Poll.** Let:

$\mathcal{A} = \{\text{the action templates on this page}\}$

$A = \{\text{all ground instances of members of } \mathcal{A}\}$

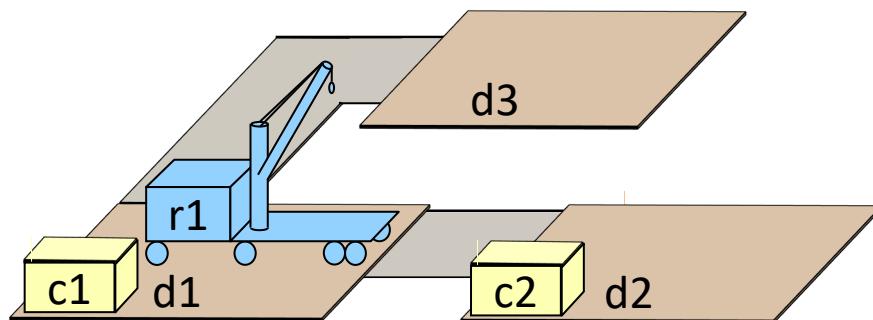
How many move actions in  $A$ ?

**Answers:**

- |      |          |
|------|----------|
| A. 1 | F. 6     |
| B. 2 | G. 7     |
| C. 3 | H. 8     |
| D. 4 | I. 9     |
| E. 5 | J. other |

# Applicability

- $a$  is *applicable* in  $s$  if
  - ▶ for every positive literal  $l \in \text{pre}(a)$ ,  $l \in s$  or  $l$  is in one of the rigid relations
  - ▶ for every negative literal  $\neg l \in \text{pre}(a)$ ,  $l \notin s$  and  $l$  isn't in any of the rigid relations
- Rigid relation
  - $\text{adjacent} = \{(d1,d2), (d2,d1), (d1,d3), (d3,d1)\}$
- *State*
  - $s_1 = \{\text{loc}(r1)=d1, \text{cargo}(r1)=\text{nil}, \text{loc}(c1)=d1\}$



- Action template  
 $\text{move}(r,l,m)$   
pre:  $\text{loc}(r)=l$ ,  $\text{adjacent}(l, m)$   
eff:  $\text{loc}(r) \leftarrow m$   
 $\text{Range}(r) = \text{Robots}$   
 $\text{Range}(l) = \text{Range}(m) = \text{Locs}$
- Applicable:  
 $\text{move}(r1,d1,d2)$   
pre:  $\text{loc}(r1)=d1$ ,  $\text{adjacent}(d1,d2)$   
eff:  $\text{loc}(r1) \leftarrow d2$
- Not applicable:  
 $\text{move}(r1,d2,d1)$   
pre:  $\text{loc}(r1)=d2$ ,  $\text{adjacent}(d2,d1)$   
eff:  $\text{loc}(r1) \leftarrow d1$

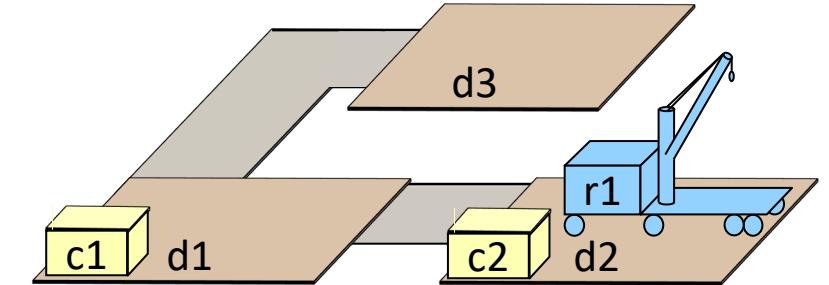
- Poll:** How many move actions are applicable in  $s_1$ ?
- |      |          |
|------|----------|
| A. 1 | F. 6     |
| B. 2 | G. 7     |
| C. 3 | H. 8     |
| D. 4 | I. 9     |
| E. 5 | J. other |

# State-Transition Function

- If  $a$  is applicable in  $s$ :

- ▶  $\gamma(s, a) = \{\text{every literal in } s \text{ that isn't changed in } \text{eff}(a)\}$   
 $\cup \{\text{every literal in } \text{eff}(a)\}$

- $s_2 = \{\text{loc}(r1)=d2, \text{ cargo}(r1)=\text{nil}, \text{ loc}(c1)=d1, \text{ loc}(c2)=d2\}$



- $a = \text{take}(r1, d2, c2)$

- pre:  $\text{cargo}(r1)=\text{nil}, \text{ loc}(r1)=d2, \text{ loc}(c2)=d2$

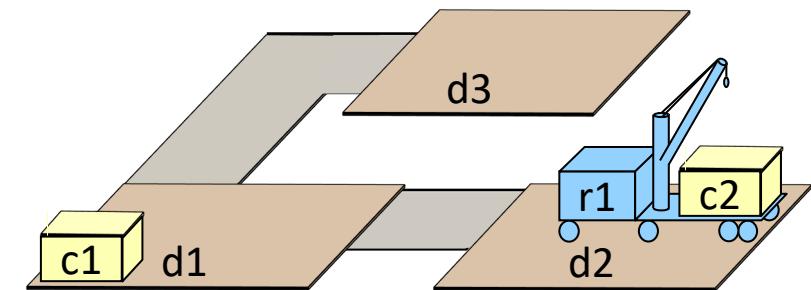
- eff:  $\text{cargo}(r1) \leftarrow c2, \text{ loc}(c2) \leftarrow r1$

- $\gamma(s_2, \text{take}(r1, d2, c2)) =$

- $\{\text{loc}(r1)=d2, \text{ loc}(c1)=d1, \text{ cargo}(r1)=c2, \text{ loc}(c2)=r1\}$

from  $s_2$

from  $\text{eff}(a)$



# State-Variable Planning Domain

- Let

$B$  = finite set of objects

$R$  = finite set of rigid relations over  $B$

$X$  = finite set of state variables

- for every state variable  $x$ ,  $\text{Range}(x) \subseteq B$

$S$  = state space over  $X$

= {all value-assignment functions that have sensible interpretations}

$\mathcal{A}$  = finite set of action templates

- for every parameter  $y$ ,  $\text{Range}(y) \subseteq B$

$A$  = {all ground instances of action templates in  $\mathcal{A}$ }

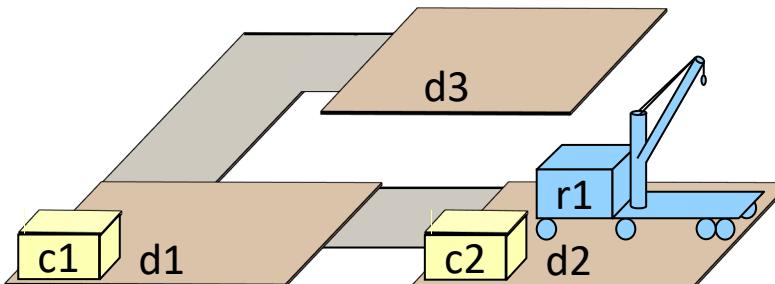
$\gamma(s,a) = \{(x,w) \mid \text{eff}(a) \text{ contains the effect } x \leftarrow w\}$

$\cup \{(x,w) \in s \mid x \text{ isn't the target of any effect in } \text{eff}(a)\}$



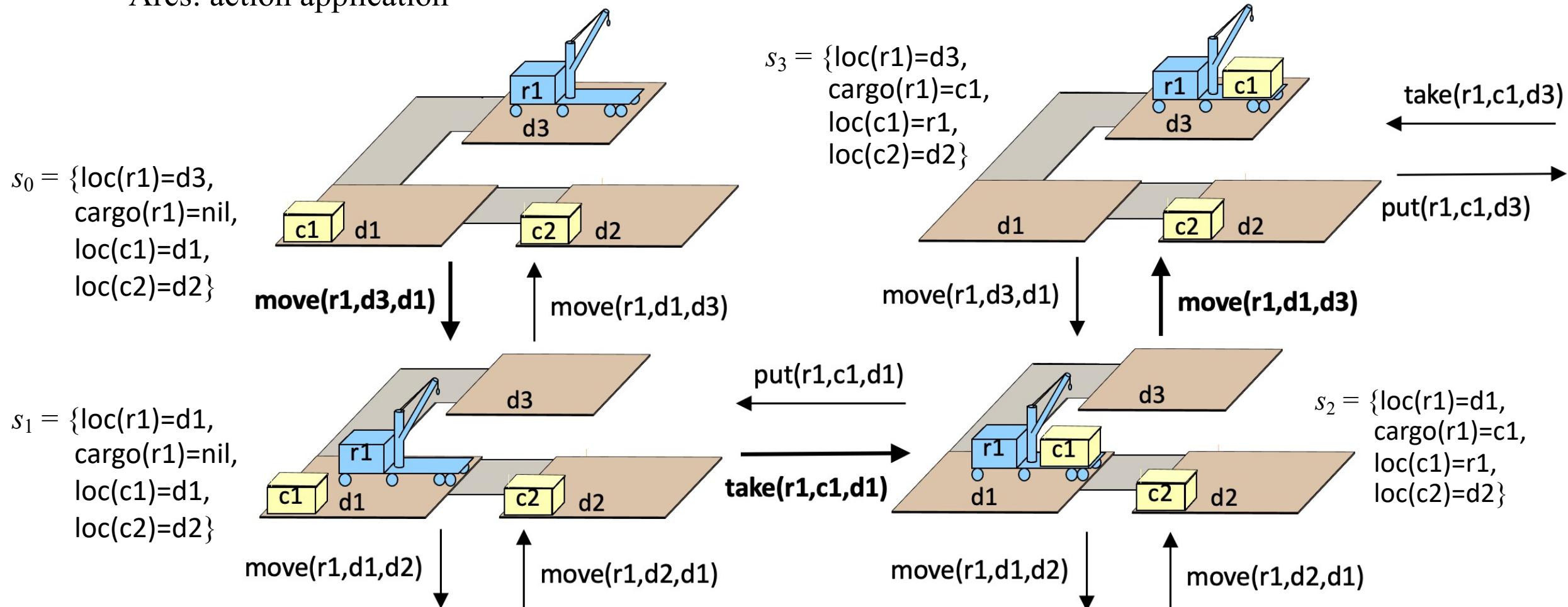
- Then  $\Sigma = (S, A, \gamma)$  is a *state-variable planning domain*

# Interpretations

- Let  $s$  be a value-assignment function
    - $s$  is a state only if the values make sense in the planning domain we're trying to represent
      - (relation to *model theory*)
  - Can  $\text{loc}(c1)=r1$  if  $\text{cargo}(r1)=\text{nil}$ ?
    - Not in our intended *interpretation*
      - Mapping of symbols to what they represent
  - Can both  $\text{loc}(c1)=r1$  and  $\text{loc}(c2)=r1$ ?
    - In our intended interpretation, can a robot carry more than one object at a time?
  - How to enforce the intended interpretation?
    - Explicitly
      - Mathematical axioms
      - Integrity constraints
    - Implicitly
      - Write an initial state  $s_0$  that satisfies the interpretation
      - Write the actions in such a way that whenever  $s$  satisfies the interpretation,  $\gamma(s,a)$  will too
- $s_2 = \{\text{loc}(r1)=d2,$   
 $\text{cargo}(r1)=\text{nil},$   
 $\text{loc}(c1)=d1,$   
 $\text{loc}(c2)=d2\}$
- 
- The diagram illustrates a planning domain with several components: two rectangular blocks labeled  $c1$  and  $c2$  located on platforms  $d1$  and  $d2$  respectively; a blue rectangular block labeled  $r1$  representing a robot, which is currently carrying  $c1$  and positioned above platform  $d2$ ; and three rectangular platforms labeled  $d1$ ,  $d2$ , and  $d3$ . Platform  $d1$  is on the left,  $d2$  is in the middle, and  $d3$  is on the right.
- ```
graph LR; s0[s0] -- a1 --> s1["s1 = γ(s0, a1)"]; s0 -- a2 --> s2["s2 = γ(s0, a2)"]; s0 -- a3 --> s3["s3 = γ(s0, a3)"]; s1 --> ...; s2 --> ...; s3 --> ...;
```
- A sequence of states  $s_0, s_1, s_2, s_3$  is shown, where each state is the result of applying an action  $a_i$  to the previous state  $s_{i-1}$ . The states are connected by red arrows, with ellipses indicating continuation. The labels for the states include the formula  $\gamma(s_0, a_i)$ .

# State Space

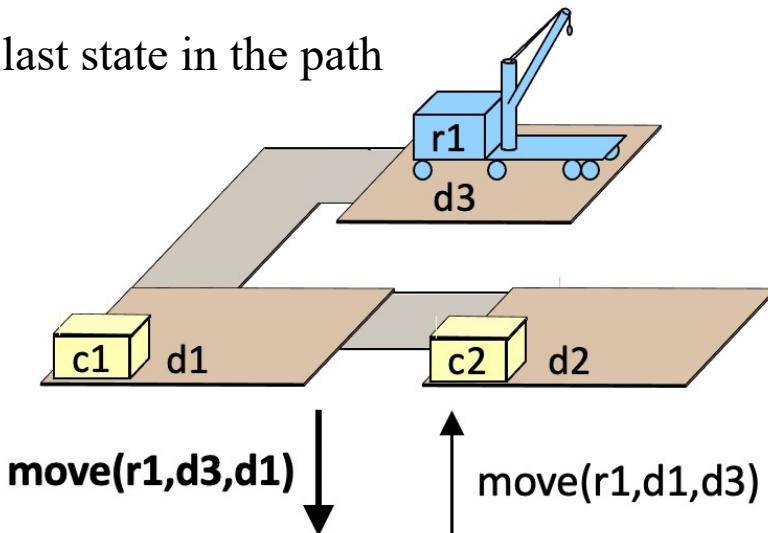
- *State Space*: a directed graph
  - ▶ Nodes = states of the world
  - ▶ Arcs: action application



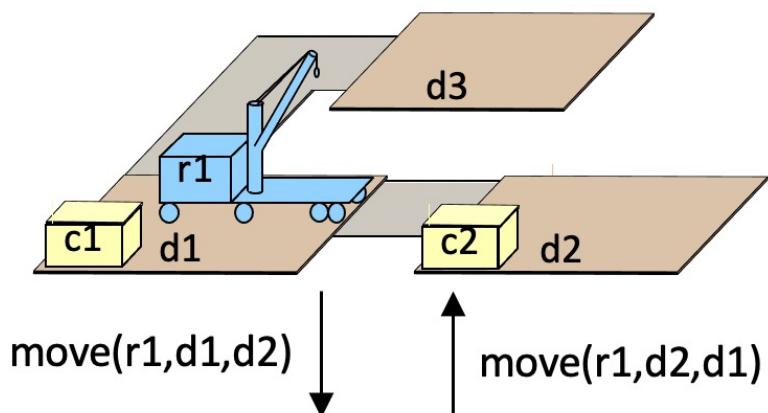
# Applying a Plan

- A plan  $\pi$  is applicable in a state  $s$  if we can apply the actions in the order that they appear in  $\pi$
- This produces a path in the state space
- Let  $\gamma(s, \pi) =$  the last state in the path

$s_0 = \{\text{loc}(r1)=d3, \text{cargo}(r1)=\text{nil}, \text{loc}(c1)=d1, \text{loc}(c2)=d2\}$

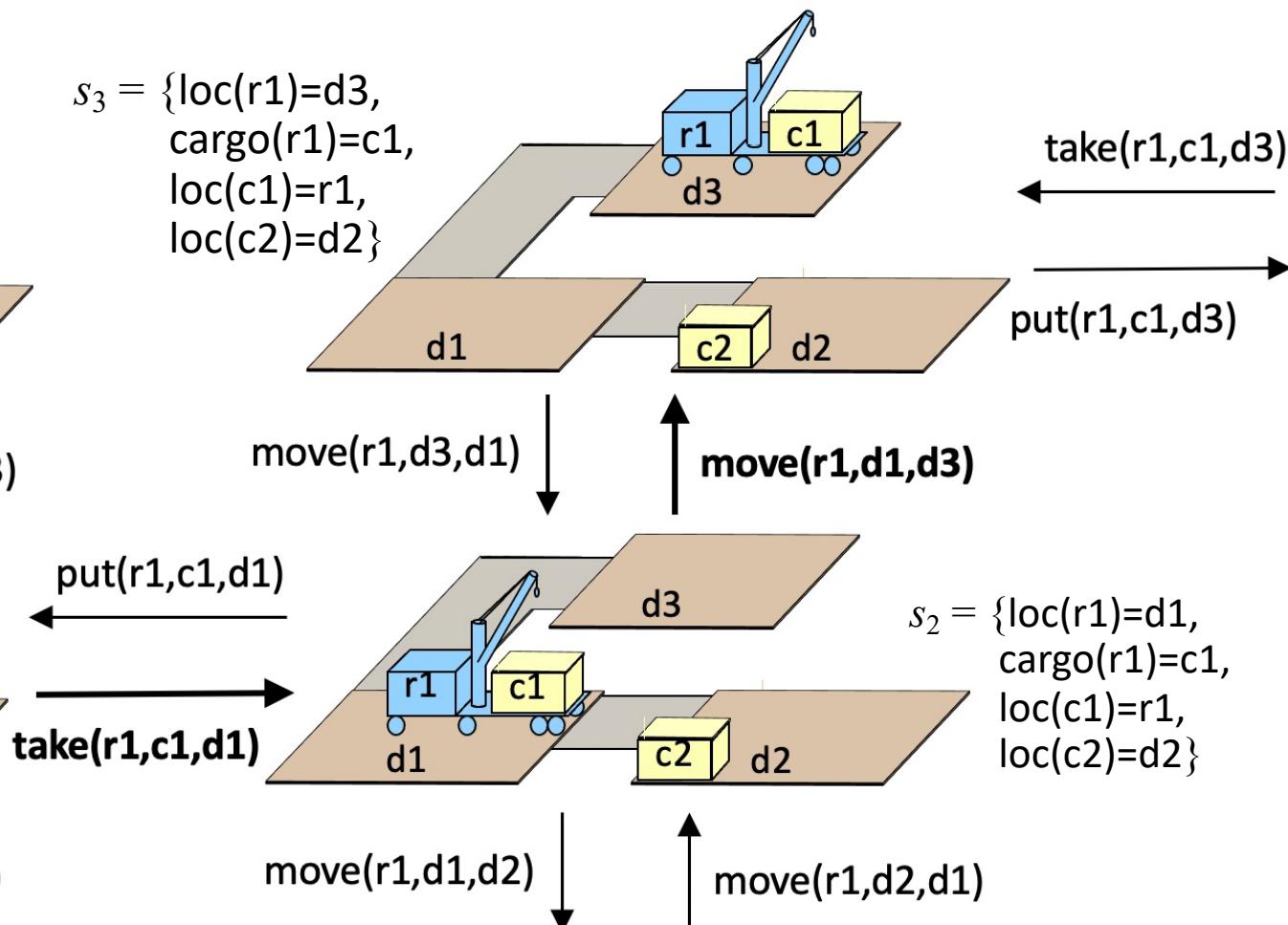


$s_1 = \{\text{loc}(r1)=d1, \text{cargo}(r1)=\text{nil}, \text{loc}(c1)=d1, \text{loc}(c2)=d2\}$



- If  $\pi = \langle \text{move}(r1, d3, d1), \text{take}(r1, d1, c1), \text{move}(r1, d1, d3) \rangle$   
then  $\gamma(s_0, \pi) = s_3$

$s_3 = \{\text{loc}(r1)=d3, \text{cargo}(r1)=c1, \text{loc}(c1)=r1, \text{loc}(c2)=d2\}$



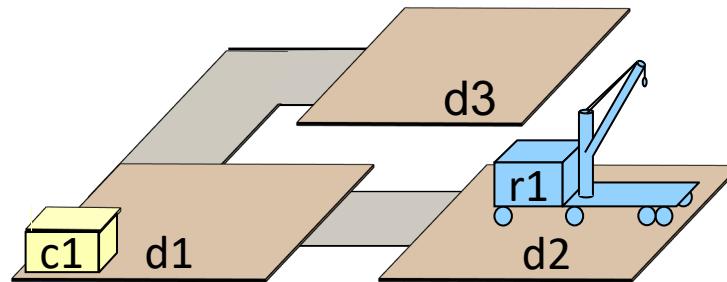
# Planning Problems

- *State-variable planning problem*: a triple  $P = (\Sigma, s_0, g)$ , where
  - ▶  $\Sigma = (S, A, \gamma)$  is a state-variable planning domain
  - ▶  $s_0 \in S$  is the *initial state*
  - ▶  $g$  is a set of ground literals called the *goal*
- $S_g = \{\text{all states in } S \text{ that satisfy } g\}$   
 $= \{s \in S \mid s \cup R \text{ contains every positive literal in } g, \text{ and none of the negative literals in } g\}$
- If  $\gamma(s_0, \pi)$  satisfies  $g$  (or equivalently,  $\gamma(s_0, \pi) \in S_g$ ) then  $\pi$  is a *solution* for  $P$

**Poll:** How many solutions of length 3?

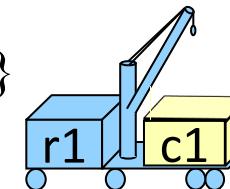
- A. 1      B. 2      C. 3      D. 4      E. 5  
 F. 6      G. 7      H. 8      I. 9      J. other

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



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

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



$\langle \text{move}(r1, d2, d1), \text{take}(r1, d1, c1) \rangle$   
 is a solution of length 2

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

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

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

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

# Classical Representation

- Motivation
  - ▶ The field of AI planning started out as automated theorem proving
  - ▶ It still uses a lot of that notation
- Classical representation is equivalent to state-variable representation
  - ▶ No distinction between rigid and varying properties
  - ▶ Both represented as logical predicates
  - ▶ Both are in the current state

$\text{adjacent}(l,m)$

- location  $l$  is adjacent to  $m$

$\text{loc}(r) = l \rightarrow \text{loc}(r;l)$

- robot  $r$  is at location  $l$

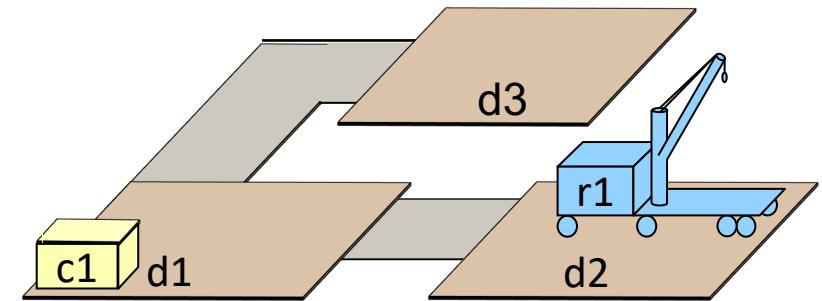
$\text{loc}(c) = r \rightarrow \text{loc}(c,r)$

- container  $c$  is on robot  $r$

$\text{cargo}(r) = c \rightarrow \text{loaded}(r)$

- there's a container on  $r$

why not  $\text{loaded}(r,c)$ ?



- State  $s$  = a set of ground atoms
  - ▶ Atom  $a$  is true in  $s$  iff  $a \in s$

$s_0 = \{\text{adjacent}(d1,d2), \text{adjacent}(d2,d1), \text{adjacent}(d1,d3), \text{adjacent}(d3,d1), \text{loc}(c1,d1), \text{loc}(r1,d2)\}$

**Poll:** Should  $s_0$  also contain  
 $\neg \text{loaded}(r1)$ ?

- A: yes    B: no  
C: unsure

# Classical planning operators

- Action templates

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

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

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

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

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

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

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

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

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

- Classical planning operators

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

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

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

pre:  $\neg\text{loaded}(r)$ ,  $\text{loc}(r,l)$ ,  $\text{loc}(c,l)$   
eff:  $\text{loaded}(r)$ ,  $\neg\text{loc}(c,l)$ ,  $\text{loc}(c,r)$

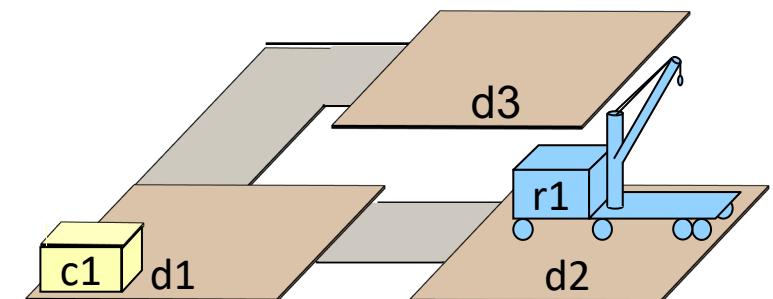
$\text{put}(r,l,c)$

pre:  $\text{loc}(r,l)$ ,  $\text{loc}(c,r)$   
eff:  $\neg\text{loaded}(r)$ ,  $\text{loc}(c,l)$ ,  $\neg\text{loc}(c,r)$

**Poll:** Does move really need to include  $\neg\text{loc}(r,l)$ ?

A: yes    B: no

C: unsure



# Actions

\*explained previously\*

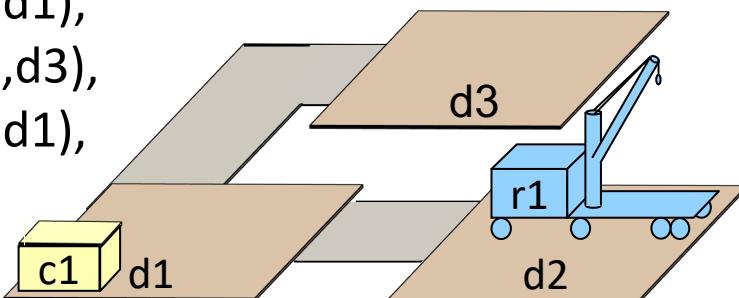
- Planning operator:

$o:$  move( $r,l,m$ )  
 pre: loc( $r,l$ ), adjacent( $l,m$ )  
 eff:  $\neg$ loc( $r,l$ ), loc( $r,m$ )

- Action:

$a_1:$  move( $r_1,d_2,d_1$ )  
 pre: loc( $r_1,d_2$ ), adjacent( $d_2,d_1$ )  
 eff:  $\neg$ loc( $r_1,d_2$ ), loc( $r_1,d_1$ )

$s_0 = \{\text{adjacent}(d_1,d_2),$   
 $\text{adjacent}(d_2,d_1),$   
 $\text{adjacent}(d_1,d_3),$   
 $\text{adjacent}(d_3,d_1),$   
 $\text{loc}(c_1,d_1),$   
 $\text{loc}(r_1,d_2)\}$



- Let

►  $\text{pre}^-(a) = \{a\text{'s negated preconditions}\}$   
 ►  $\text{pre}^+(a) = \{a\text{'s non-negated preconditions}\}$

- $a$  is applicable in state  $s$  iff

$s \cap \text{pre}^-(a) = \emptyset$  and  $\text{pre}^+(a) \subseteq s$

if state does not include the negated preconditions of action  $a$ , but should include the non-negated ones

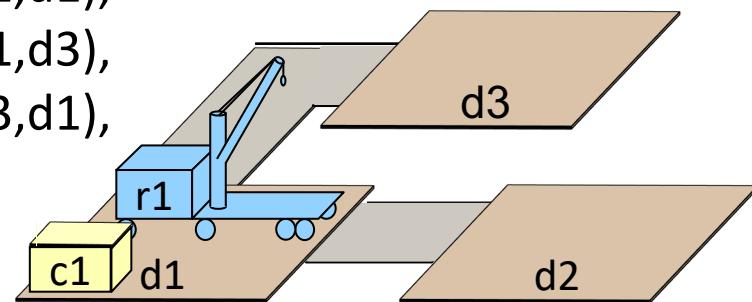
- If  $a$  is applicable in  $s$  then

►  $\gamma(s,a) = (s \setminus \text{eff}^-(a)) \cup \text{eff}^+(a)$

means remove from State transition the negated effects & add the non-negated ones

meaning?

$\gamma(s_0, a_1) = \{\text{adjacent}(d_1,d_2),$   
 $\text{adjacent}(d_2,d_1),$   
 $\text{adjacent}(d_1,d_3),$   
 $\text{adjacent}(d_3,d_1),$   
 $\text{loc}(c_1,d_1),$   
 $\text{loc}(r_1,d_1)\}$



# Discussion

$$x(b_1, \dots, b_{n-1}) = b_n \Rightarrow P_x(b_1, \dots, b_{n-1}, b_n)$$

State-variable  
rep.

Classical  
rep.

$$x_P(b_1, \dots, b_k) = 1 \Leftarrow P(b_1, \dots, b_k)$$

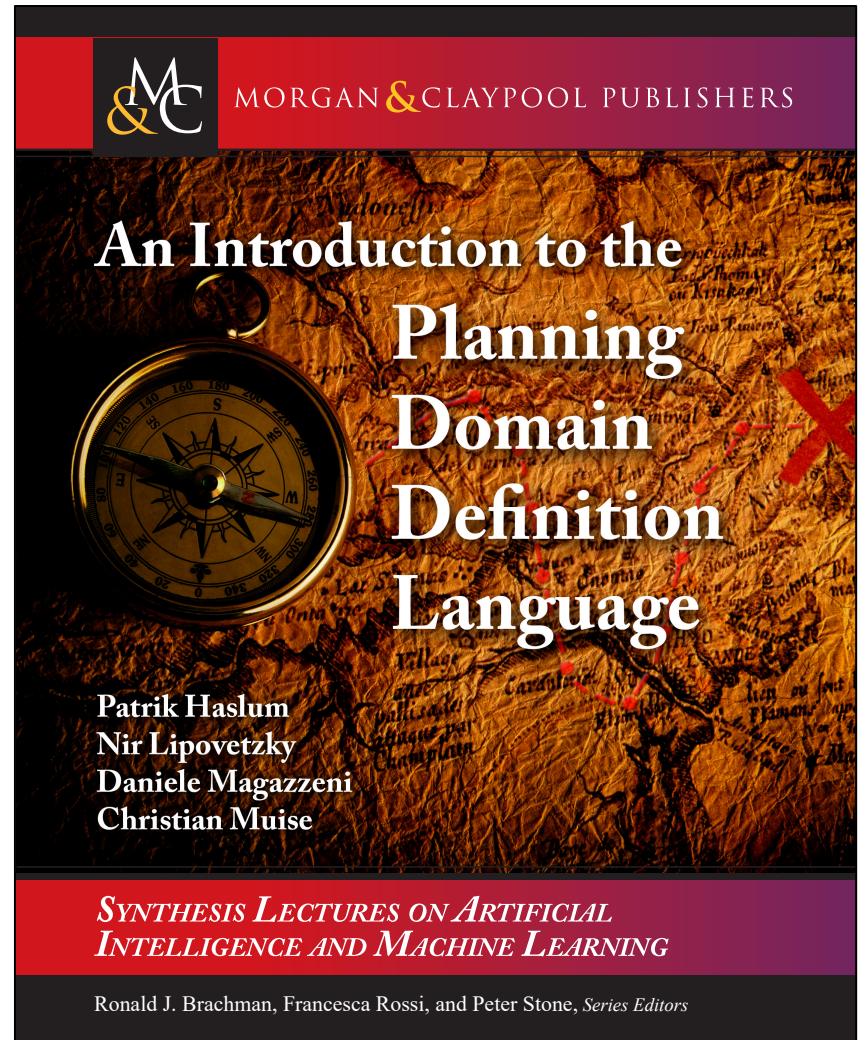
- Equivalent expressive power
  - ▶ Each can be converted to the other in linear time and space
- Classical representation
  - ▶ More natural for logicians
  - ▶ Don't require single-valued functions
- State variables
  - ▶ More natural for engineers and computer programmers
  - ▶ When changing a value, don't have to explicitly delete the old one
- Historically, classical representation has been more widely used
  - ▶ That's starting to change

**Poll:** Could we instead use  
 $x_P(b_1, \dots, b_{k-1}) = b_k$ ?

A: yes    B: no  
C: unsure

# PDDL

- Language for defining planning domains and problems
- Original version of PDDL ≈ 1996
  - ▶ Just classical planning
- Multiple revisions and extensions
  - ▶ Different subsets accommodate different kinds of planning
- We'll discuss the classical-planning subset
  - ▶ Chapter 2 of the PDDL book



# Example domain

- Classical representation:

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

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

Effects:  $\neg\text{loc}(r,l)$ ,  $\text{loc}(r,m)$

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

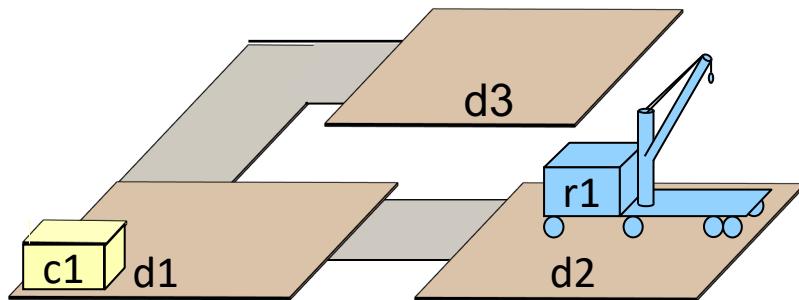
Precond:  $\text{loc}(r,l)$ ,  $\text{loc}(c,l)$ ,  $\neg\text{loaded}(r)$

Effects:  $\text{loc}(c,r)$ ,  $\neg\text{loc}(c,l)$ ,  $\text{loaded}(r)$

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

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

Effects:  $\text{loc}(c,l)$ ,  $\neg\text{loc}(c,r)$ ,  $\neg\text{loaded}(r)$



```
(define (domain example-domain-1)
  (requirements :negative-preconditions)

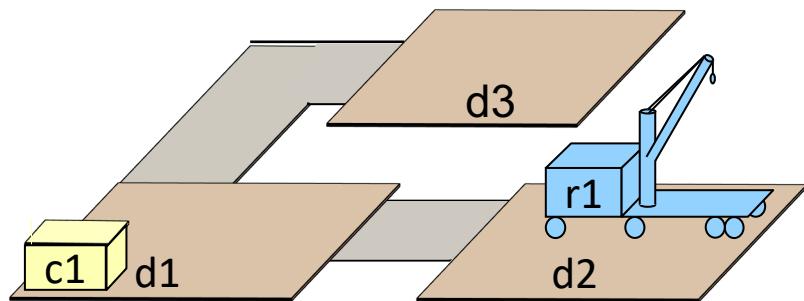
  (:action move
    :parameters (?r ?l ?m)
    :precondition (and (loc ?r ?l)
                        (adjacent ?l ?m))
    :effect (and (not (loc ?r ?l))
                  (loc ?r ?m)))

  (:action take
    :parameters (?r ?l ?c)
    :precondition (and (loc ?r ?l)
                        (loc ?c ?l)
                        (not (loaded ?r)))
    :effect (and (not (loc ?c ?l))
                  (loc ?c ?r)
                  (loaded ?r)))

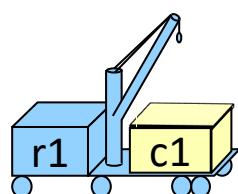
  (:action put
    :parameters (?r ?l ?c)
    :precondition (and (loc ?r ?l)
                        (loc ?c ?r))
    :effect (and (loc ?c ?l)
                  (not (loc ?c ?r))
                  (not (loaded ?r))))))
```

# Example problem

- Classical representation:



$s_0 = \{\text{adjacent}(d1, d2), \text{adjacent}(d2, d1), \text{adjacent}(d1, d3), \text{adjacent}(d3, d1), \text{loc}(c1, d1), \text{loc}(r1, d2)\}$



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

```
(define (problem example-problem-1)
  (:domain example-domain-1))
```

```
(:init
  (adjacent d1 d2)
  (adjacent d2 d1)
  (adjacent d1 d3)
  (adjacent d3 d1)
  (loc c1 d1)
  (loc r1 d2))
```

```
(:goal (loc c1 r1)))
```

# Typed domain

State-variable planning:

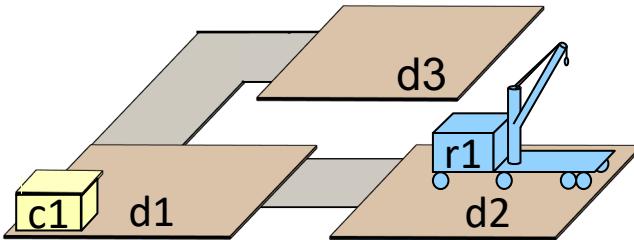
- Sets of objects
  - ▶  $B = \text{Movable\_objects} \cup \text{Locs}$
  - ▶  $\text{Movable\_objects} = \text{Robots} \cup \text{Containers}$
  - ▶  $\text{Robots} = \{r_1\}$
  - ▶  $\text{Containers} = \{c_1\}$
  - ▶  $\text{Locs} = \{d_1, d_2, d_3\}$

```
(define (domain example-domain-2)
```

```
  (:requirements
    :negative-preconditions
    :typing)
```

```
  (:types
    location movable-obj - object
    robot container - movable-obj)]
```

```
  (:predicates
    (loc ?r - movable-obj
        ?l - location)
    (loaded ?r - robot)
    (adjacent ?l ?m - location))
```



- Parameter ranges
  - ▶  $r \in \text{Robots}$
  - ▶  $l, m \in \text{Locs}$
  - ▶  $c \in \text{Containers}$

$\text{Locations}, \text{Movable\_objects} \subseteq B$   
 $\text{Robots}, \text{Containers} \subseteq \text{Movable\_objects}$

```
(:action move
  :parameters (?r - robot
               ?l ?m - location)
  :precondition (and (loc ?r ?l)
                      (adjacent ?l ?m))
  :effect (and (not (loc ?r ?l))
                (loc ?r ?m)))
```

```
(:action take
  :parameters (?r - robot
               ?l - location
               ?c - container)
  :precondition (and (loc ?r ?l)
                      (loc ?c ?l)
                      (not (loaded ?r)))
  :effect (and (not (loc ?r ?l))
                (loc ?r ?m)))
```

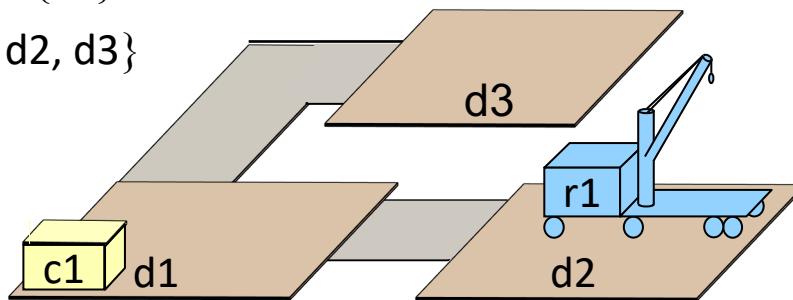
```
(:action put
  :parameters (?r - robot
               ?l - location
               ?c - container)
  :precondition (and (loc ?r ?l)
                      (loc ?c ?r))
  :effect (and (loc ?c ?l)
                (not (loc ?c ?r))
                (not (loaded ?r)))))
```

like saying  $r \in \text{Robots}$ ,  
 $l \in \text{Locs}$ ,  
 $c \in \text{Containers}$

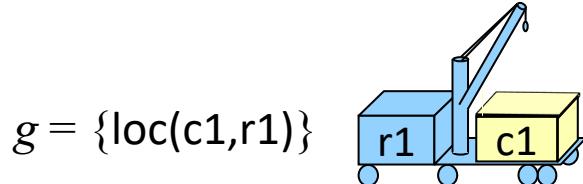
# Typed problem

State-variable planning:

- Sets of objects
  - ▶  $B = \text{Movable\_objects} \cup \text{Locs}$
  - ▶  $\text{Movable\_objects} = \text{Robots} \cup \text{Containers}$
  - ▶  $\text{Robots} = \{r1\}$
  - ▶  $\text{Containers} = \{c1\}$
  - ▶  $\text{Locs} = \{d1, d2, d3\}$



$s_0 = \{\text{adjacent}(d1, d2), \text{adjacent}(d2, d1), \text{adjacent}(d1, d3), \text{adjacent}(d3, d1), \text{loc}(c1, d1), \text{loc}(r1, d2)\}$



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

```
(define (problem example-problem-2)
  (:domain example-domain-2))

  (:objects
    r1 - robot
    c1 - container
    d1 d2 d3 - location)

  (:init
    (adjacent d1 d2)
    (adjacent d2 d1)
    (adjacent d1 d3)
    (adjacent d3 d1)
    (loc c1 d1)
    (loc r1 d2))

  (:goal (loc c1 r1)))
```

# Summary

Section 2.1 of Ghallab *et al.* (2016)

- State-Variable Representation
  - ▶ State-transition systems, classical planning assumptions
  - ▶ Classical planning problems, plans, solutions
  - ▶ Objects, rigid properties
  - ▶ Varying properties, state variables, states as functions
  - ▶ Action templates, actions, applicability,  $\gamma$
  - ▶ State-variable planning domains, plans, problems, solutions
  - ▶ Comparison with classical representation

Chapter 2 of Haslum *et al.* (2019)

- Classical fragment of PDDL
  - ▶ Planning domains, planning problems
  - ▶ untyped, typed

# Outline

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

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

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

- 2.3 Heuristic functions
  - 2.7.7 HTN planning
- 

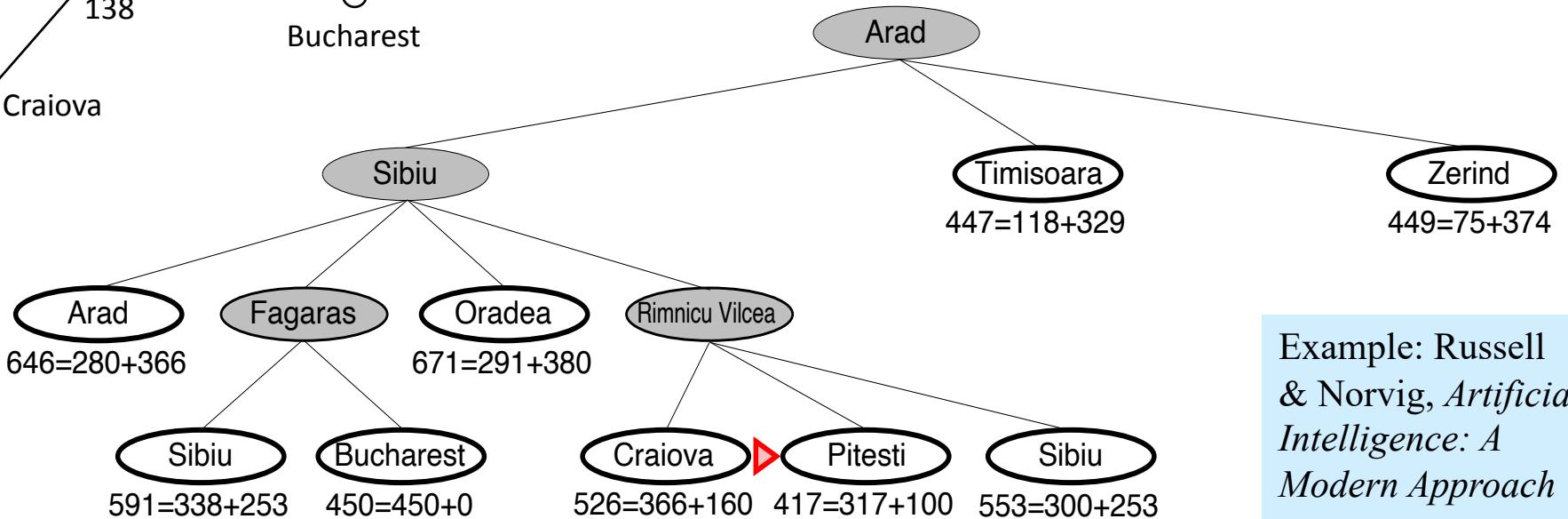
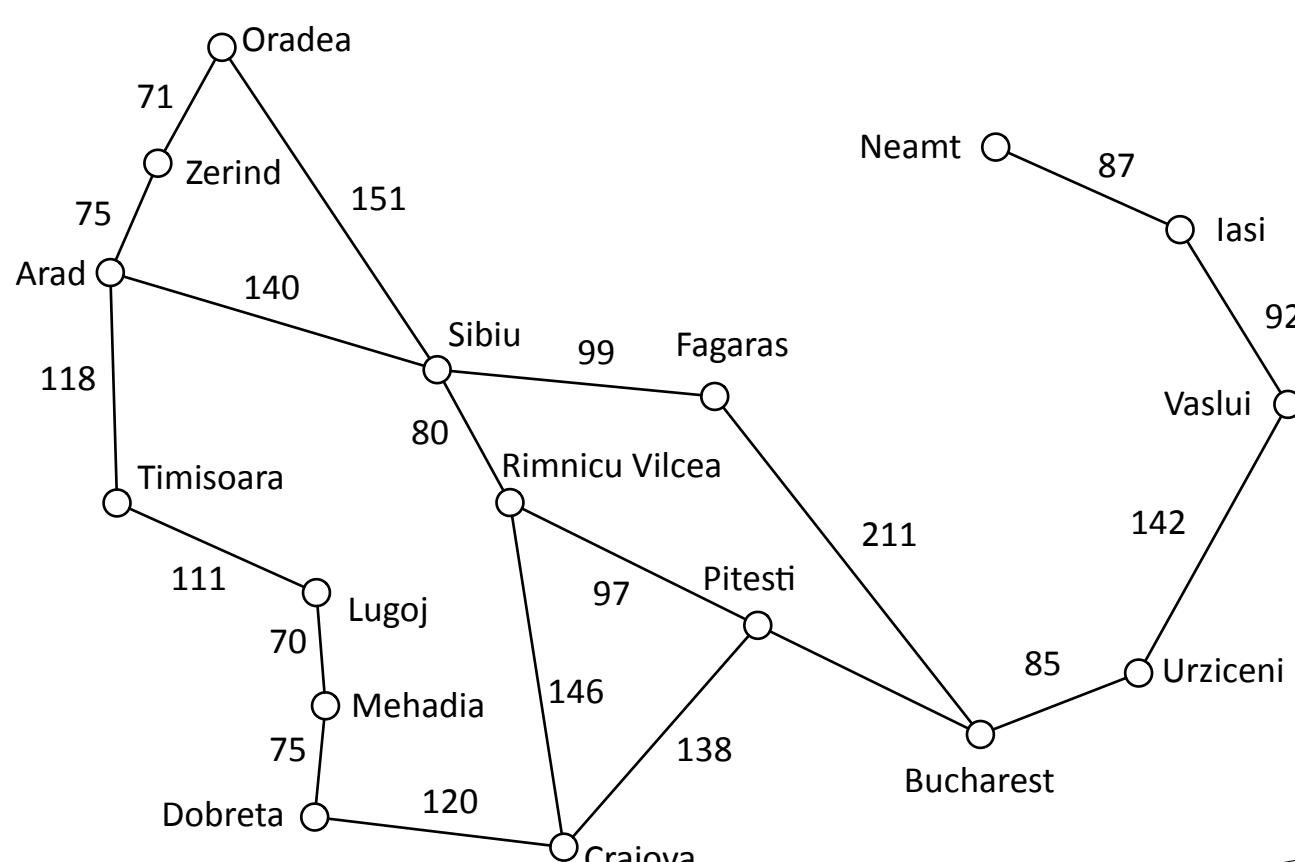
Chapter 2, part *c* (chap2c.pdf):

- 2.4 Backward search
  - 2.5 Plan-space search
- 

Additional slides:

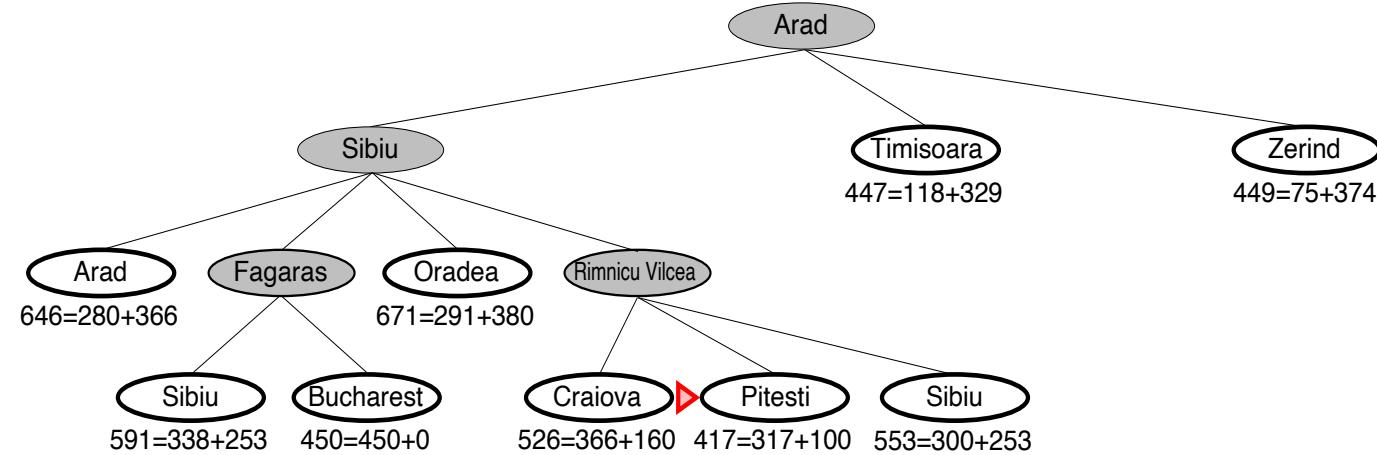
- 2.7.8 LTL\_planning.pdf

# Planning as Search



Example: Russell & Norvig, *Artificial Intelligence: A Modern Approach*

# Search-Tree Terminology



- *Node*  $\approx$  a pair  $v = (\pi, s)$ , where  $s = \gamma(s_0, \pi)$ 
  - ▶ In practice,  $v$  will contain other things too
    - $\text{depth}(v)$ ,  $\text{cost}(\pi)$ , pointers to parent and children, ...
  - ▶  $\pi$  isn't always stored explicitly, can be computed from the parent pointers
- *children* of  $v = \{(\pi.a, \gamma(s,a)) \mid a \text{ is applicable in } s\}$
- *successors or descendants* of  $v$ :  
children, children of children, etc.

- *ancestors* of  $v$   
 $= \{\text{nodes that have } v \text{ as a successor}\}$
- *initial* or *starting* or *root* node  $v_0 = (\langle \rangle, s_0)$ 
  - ▶ root of the search tree
- *path* in the search space: sequence of nodes  $\langle v_0, v_1, \dots, v_n \rangle$  such that each  $v_i$  is a child of  $v_{i-1}$
- *height* of search space  
 $= \text{length of longest acyclic path from } v_0$
- *depth* of  $v$   
 $= \text{length}(\pi) = \text{length of path from } v_0 \text{ to } v$
- *branching factor* of  $v$   
 $= \text{number of children of } v$
- *branching factor* of a search tree  
 $= \text{max branching factor of the nodes}$
- *expand*  $v$ : generate all children

# Forward Search

Forward-search ( $\Sigma, s_0, g$ )

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

loop

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

$A' \leftarrow \{a \in A \mid a \text{ is applicable in } s\}$

  if  $A' = \emptyset$  then return failure

  nondeterministically choose  $a \in A'$

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

- Nondeterministic algorithm
  - ▶ *Sound*: if an execution trace returns a plan  $\pi$ , 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
  - ▶ They'll all be sound
  - ▶ Whether they're complete depends on how you implement the nondeterministic choice
    - Which leaf node to expand next
    - Which nodes to prune from the search space

# Forward Search

Forward-search ( $\Sigma, s_0, g$ )

```
 $s \leftarrow s_0; \pi \leftarrow \langle \rangle$ 
loop
  if  $s$  satisfies  $g$  then return  $\pi$ 
   $A' \leftarrow \{a \in A \mid a \text{ is applicable in } s\}$ 
  if  $A' = \emptyset$  then return failure
  nondeterministically choose  $a \in A'$ 
   $s \leftarrow \gamma(s, a); \pi \leftarrow \pi.a$ 
```

- Nondeterministic algorithm
  - ▶ *Sound*: if an execution trace returns a plan  $\pi$ , 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

Many of the algorithms in this class:

Deterministic-Search( $\Sigma, s_0, g$ )

```
 $Frontier \leftarrow \{(\langle \rangle, s_0)\}$ 
 $Expanded \leftarrow \emptyset$ 
while  $Frontier \neq \emptyset$  do
  select a node  $v = (\pi, s) \in Frontier$  (i)
  remove  $v$  from  $Frontier$ 
  add  $v$  to  $Expanded$ 
  if  $s$  satisfies  $g$  then return  $\pi$ 
   $Children \leftarrow \{(\pi.a, \gamma(s, a)) \mid s \text{ satisfies pre}(a)\}$ 
  prune 0 or more nodes from
     $Children, Frontier, Expanded$  (ii)
   $Frontier \leftarrow Frontier \cup Children$ 
```

return failure

# Deterministic Version

Deterministic-Search( $\Sigma, s_0, g$ )

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

$Expanded \leftarrow \emptyset$

while  $Frontier \neq \emptyset$  do

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

    remove  $v$  from  $Frontier$

    add  $v$  to  $Expanded$

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

$Children \leftarrow$

$\{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies } \text{pre}(a)\}$

    prune 0 or more nodes from

$Children, Frontier, Expanded$          (ii)

$Frontier \leftarrow Frontier \cup Children$

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  $(\pi,s)$  that has an ancestor  $(\pi',s')$  such that  $s' = s$
- In classical planning problems,  $S$  is finite
  - ▶ Cycle-checking will guarantee termination

# Breadth-First Search (BFS)

Deterministic-Search( $\Sigma, s_0, g$ )

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

$Expanded \leftarrow \emptyset$

while  $Frontier \neq \emptyset$  do

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

    remove  $v$  from  $Frontier$

    add  $v$  to  $Expanded$

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

$Children \leftarrow$

$\{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies } \text{pre}(a)\}$

    prune 0 or more nodes from

$Children, Frontier, Expanded$

$Frontier \leftarrow Frontier \cup Children$

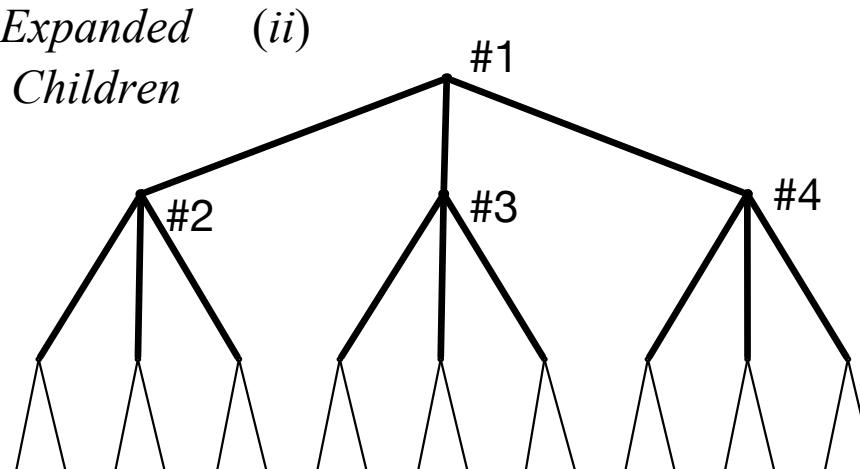
return failure

- (i): Select  $(\pi, s) \in Frontier$  that has the smallest  $\text{length}(\pi)$ , i.e., smallest number of edges
  - ▶ Tie-breaking rule: select oldest

- (ii): Remove every  $(\pi, s) \in Children \cup Frontier$  such that  $s \in Expanded$ 
  - ▶ 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  $O(b|S|)$
- ▶ where
  - $b = \text{max branching factor}$
  - $|S| = \text{number of states in } S$



# Depth-First Search (DFS)

Deterministic-Search( $\Sigma, s_0, g$ )

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

$Expanded \leftarrow \emptyset$

while  $Frontier \neq \emptyset$  do

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

    remove  $v$  from  $Frontier$

    add  $v$  to  $Expanded$

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

$Children \leftarrow$

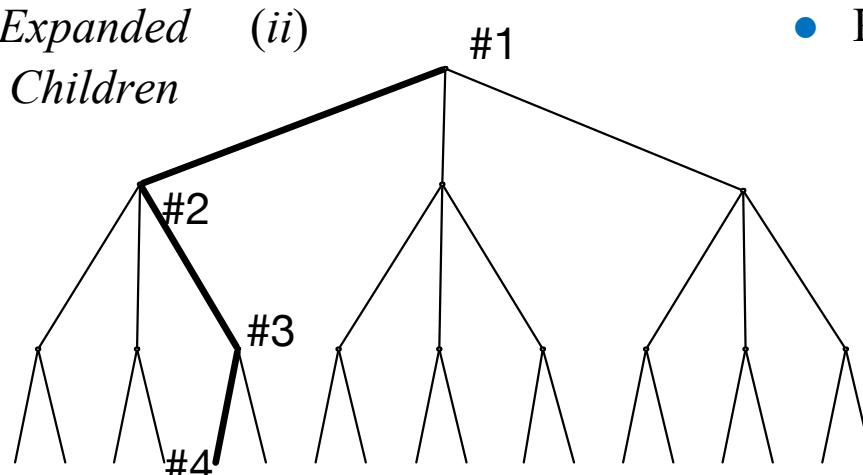
$\{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies } \text{pre}(a)\}$

    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( $\pi$ ), i.e., largest number of edges

- ▶ Possible tie-breaking rules:  
left-to-right, **smallest  $h(s)$**

- heuristic function, 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

Deterministic-Search( $\Sigma, s_0, g$ )

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

$Expanded \leftarrow \emptyset$

while  $Frontier \neq \emptyset$  do

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

    remove  $v$  from  $Frontier$

    add  $v$  to  $Expanded$

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

$Children \leftarrow$

$\{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies } \text{pre}(a)\}$

    prune 0 or more nodes from

$Children, Frontier, Expanded$

$Frontier \leftarrow Frontier \cup Children$

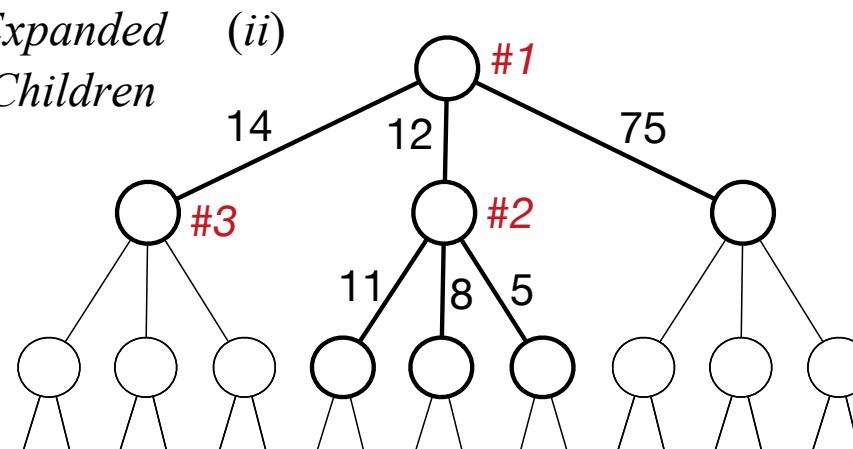
return failure

(i): Select  $(\pi, s) \in Frontier$  that has smallest cost( $\pi$ )

(ii): Prune every  $(\pi, s) \in Children \cup Frontier$   
such that  $Expanded$  already contains a node  $(\pi', s)$

- Properties

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

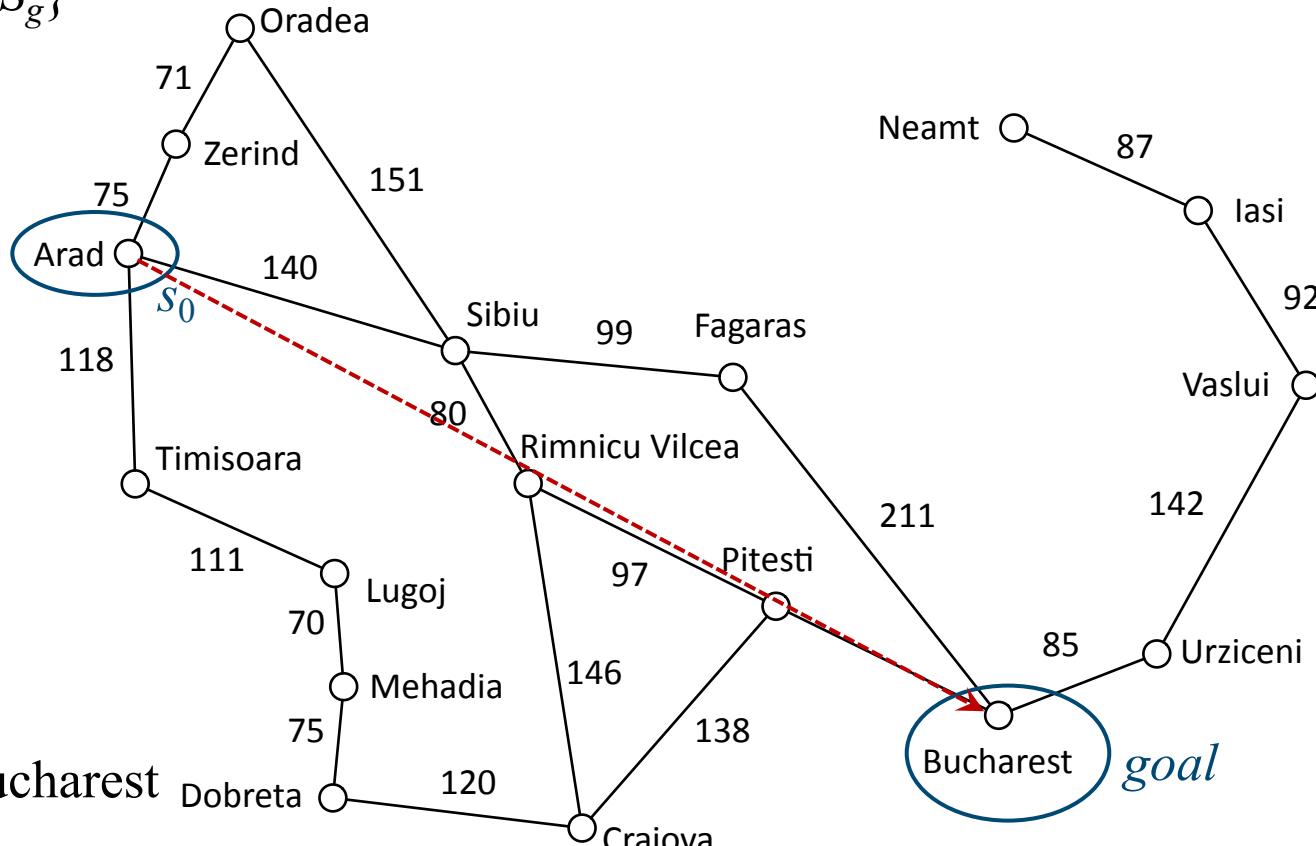


**Poll:** If node  $v$  is expanded before node  $v'$ , then how are  $\text{cost}(v)$  and  $\text{cost}(v')$  related?

- $\text{cost}(v) < \text{cost}(v')$
- $\text{cost}(v) \leq \text{cost}(v')$
- $\text{cost}(v) > \text{cost}(v')$
- $\text{cost}(v) \geq \text{cost}(v')$
- none of the above

# Heuristic Functions

- Idea: estimate the cost of getting from a state  $s$  to a goal
- Let  $h^*(s) = \min\{\text{cost}(\pi) \mid \gamma(s, \pi) \in S_g\}$ 
  - 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



from Russell & Norvig, *Artificial Intelligence: A Modern Approach*

# Greedy Best-First Search (GBFS)

Deterministic-Search( $\Sigma, s_0, g$ )

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

$Expanded \leftarrow \emptyset$

while  $Frontier \neq \emptyset$  do

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

    remove  $v$  from  $Frontier$

    add  $v$  to  $Expanded$

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

$Children \leftarrow$

$\{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies } \text{pre}(a)\}$

    prune 0 or more nodes from

$Children, Frontier, Expanded$          (ii)

$Frontier \leftarrow Frontier \cup Children$

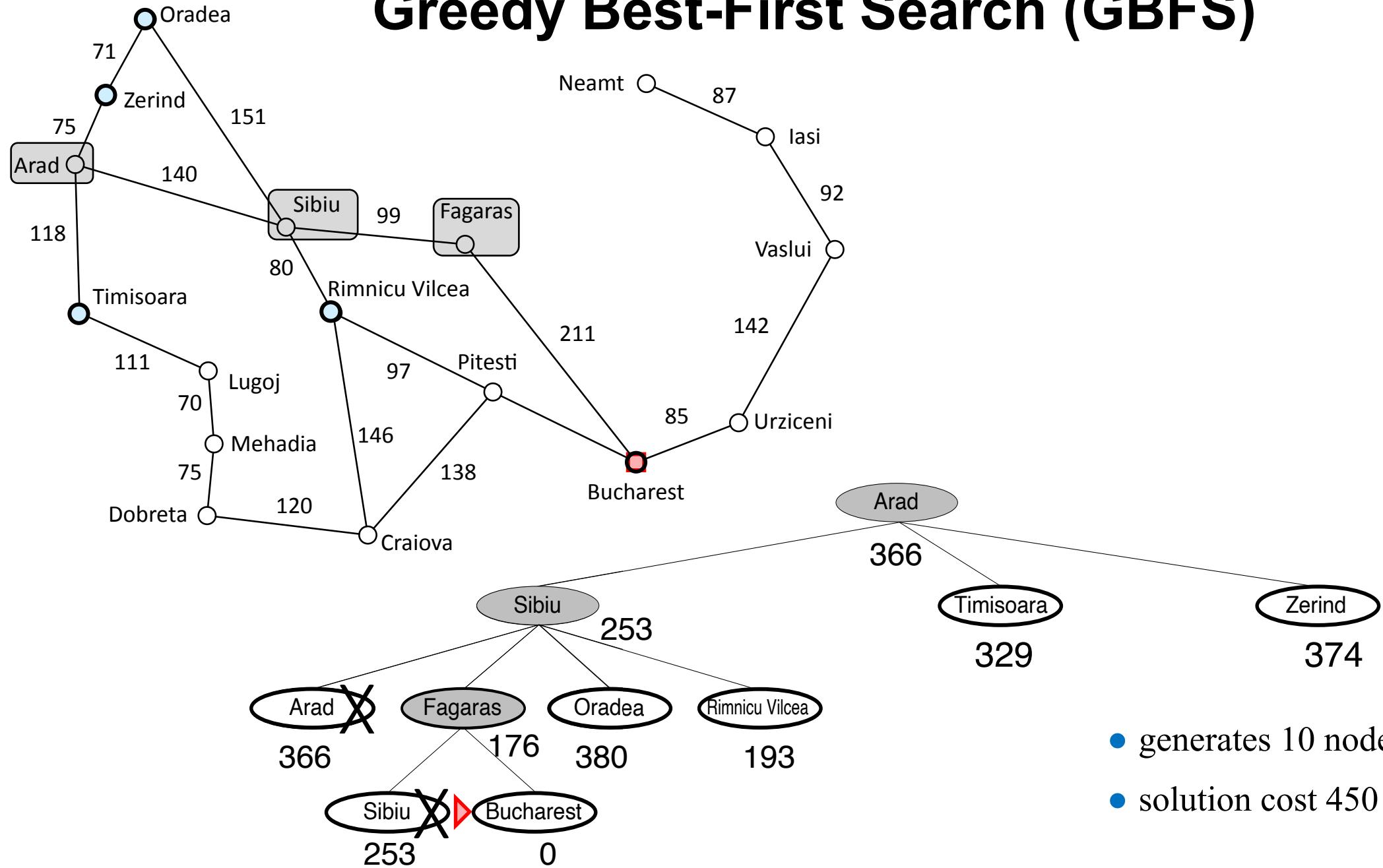
return failure

- Idea: choose a node that's likely to be close to a goal
- Node selection:
  - ▶ Select a node  $v = (\pi, s) \in Frontier$  for which  $h(s)$  is smallest
  - ▶ Tie-breaking: if more than one such node, choose the oldest
- Pruning: for every node  $v = (\pi, s)$  in  $Children$ :
  - ▶ If  $Children \cup Frontier \cup Expanded$  contains another node with state  $s$ , then we've found multiple paths from  $s_0$  to  $s$
  - ▶ Keep only the one with the lowest cost
  - ▶ If more than one such node, keep the oldest
- Properties
  - ▶ Terminates; returns a solution if one exists
  - ▶ Solution is usually found quickly, often near-optimal

**Poll:** Have you seen GBFS before?

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

# Greedy Best-First Search (GBFS)



| straight-line dist.<br>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\*

Deterministic-Search( $\Sigma, s_0, g$ )

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

$Expanded \leftarrow \emptyset$

while  $Frontier \neq \emptyset$  do

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

    remove  $v$  from  $Frontier$

    add  $v$  to  $Expanded$

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

$Children \leftarrow$

$\{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies } \text{pre}(a)\}$

    prune 0 or more nodes from

$Children, Frontier, Expanded$          (ii)

$Frontier \leftarrow Frontier \cup Children$

return failure

**Poll:** Have you seen A\* before?

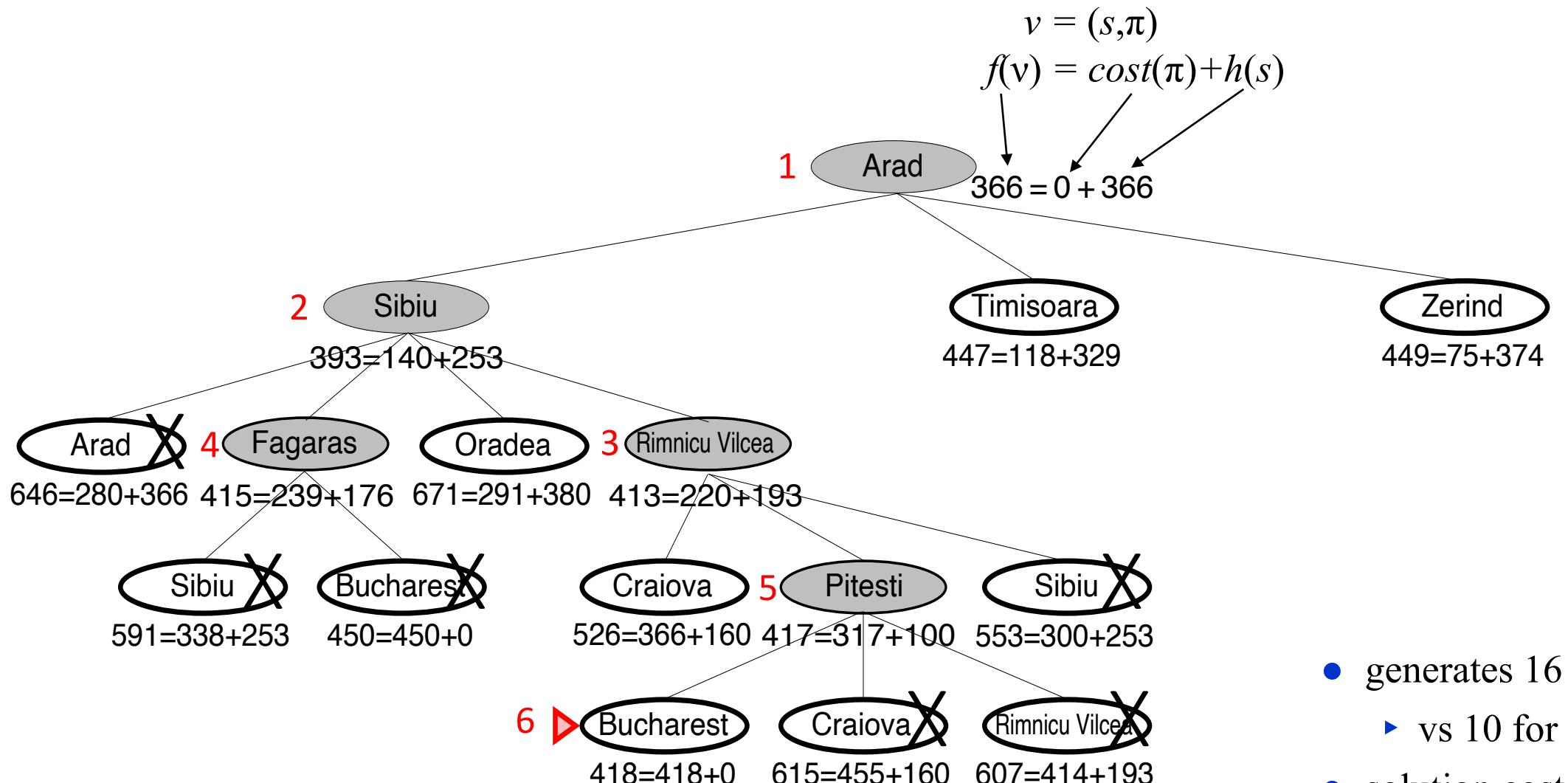
A. yes

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  $v = (\pi, s)$  in  $Frontier$  that has smallest value of  $f(v) = \text{cost}(\pi) + h(s)$ 
    - Tie-breaking rule: choose oldest
- Pruning: same as in GBFS
  - ▶ for every node  $v = (\pi, s)$  in  $Children$ :
    - If  $Children \cup Frontier \cup Expanded$  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
  - ▶ *Completeness*: returns a solution if one exists
  - ▶ *Optimality*: under certain conditions (I'll discuss later), can guarantee optimality

|                | straight-line dist.<br>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:

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

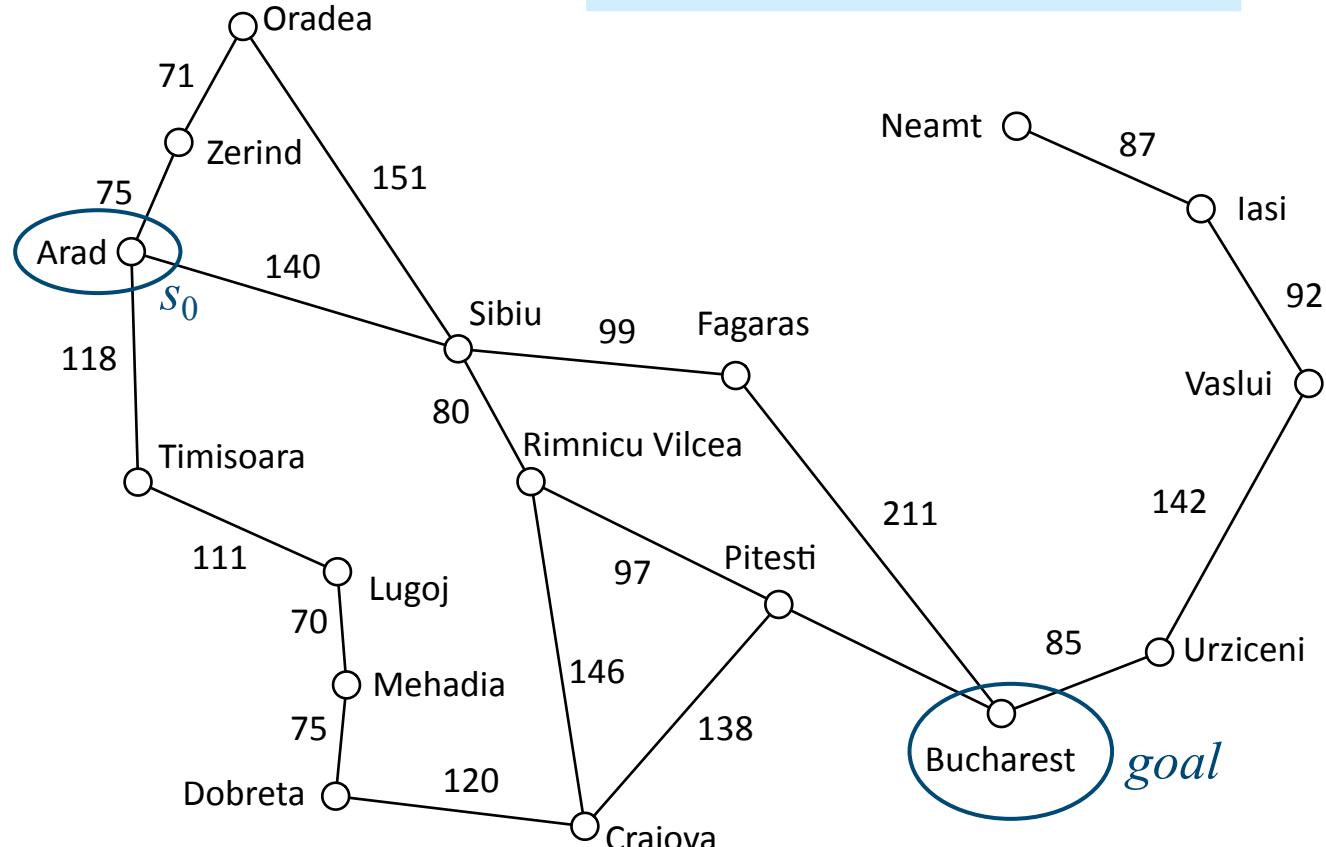
- Definition:  $h$  is *admissible* if for every  $s$ ,  $h(s) \leq h^*(s)$

- Optimality:

- in classical planning problems, if  $h$  is admissible then any solution returned by A\* will be optimal (least cost)

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

- A. Yes    B. No    C. Not sure



# Admissibility

- Notation:

- $v = (\pi, s)$ , where  $\pi$  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  $s$ ,  $h(s) \leq h^*(s)$

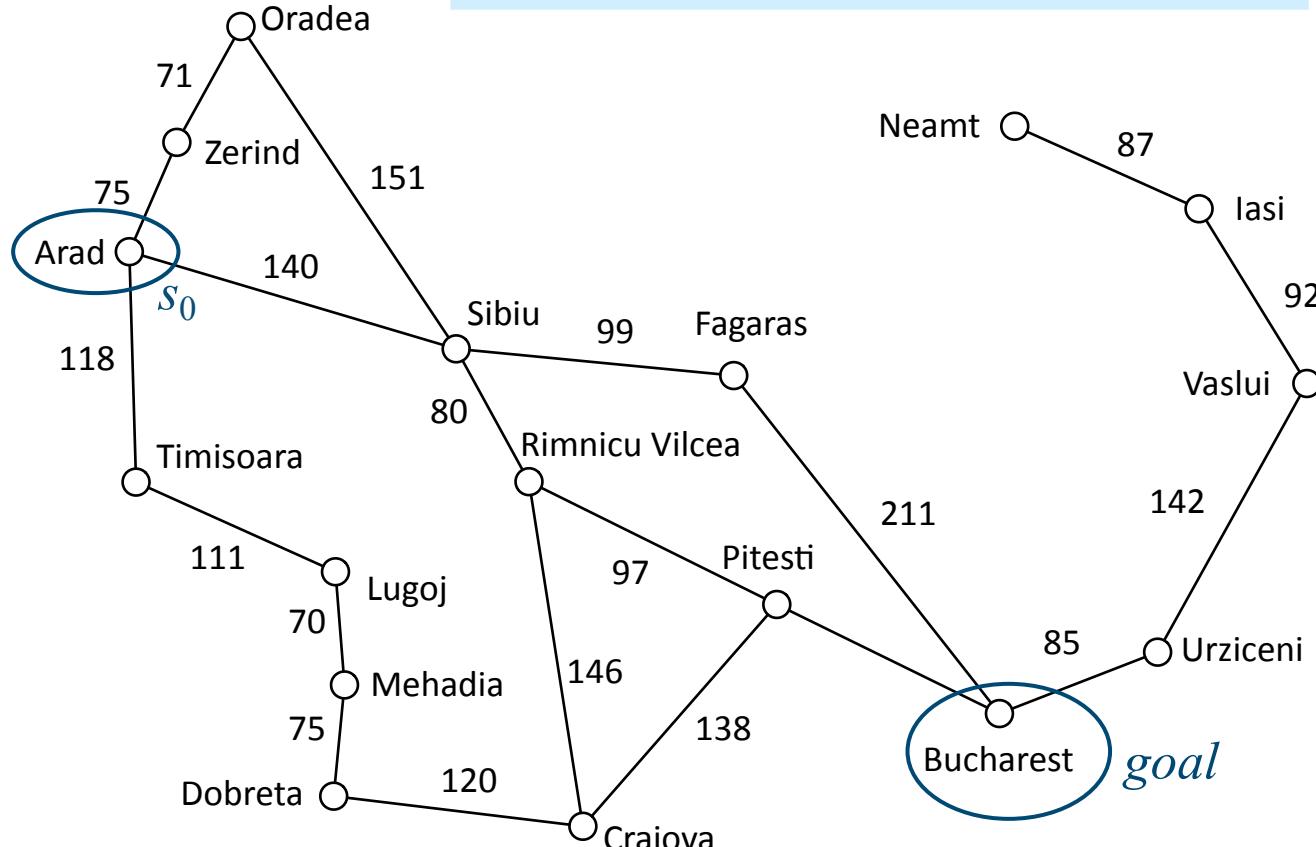
- Optimality:*

- in classical planning problems, if  $h$  is admissible then any 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



# Dominance

- Definition:

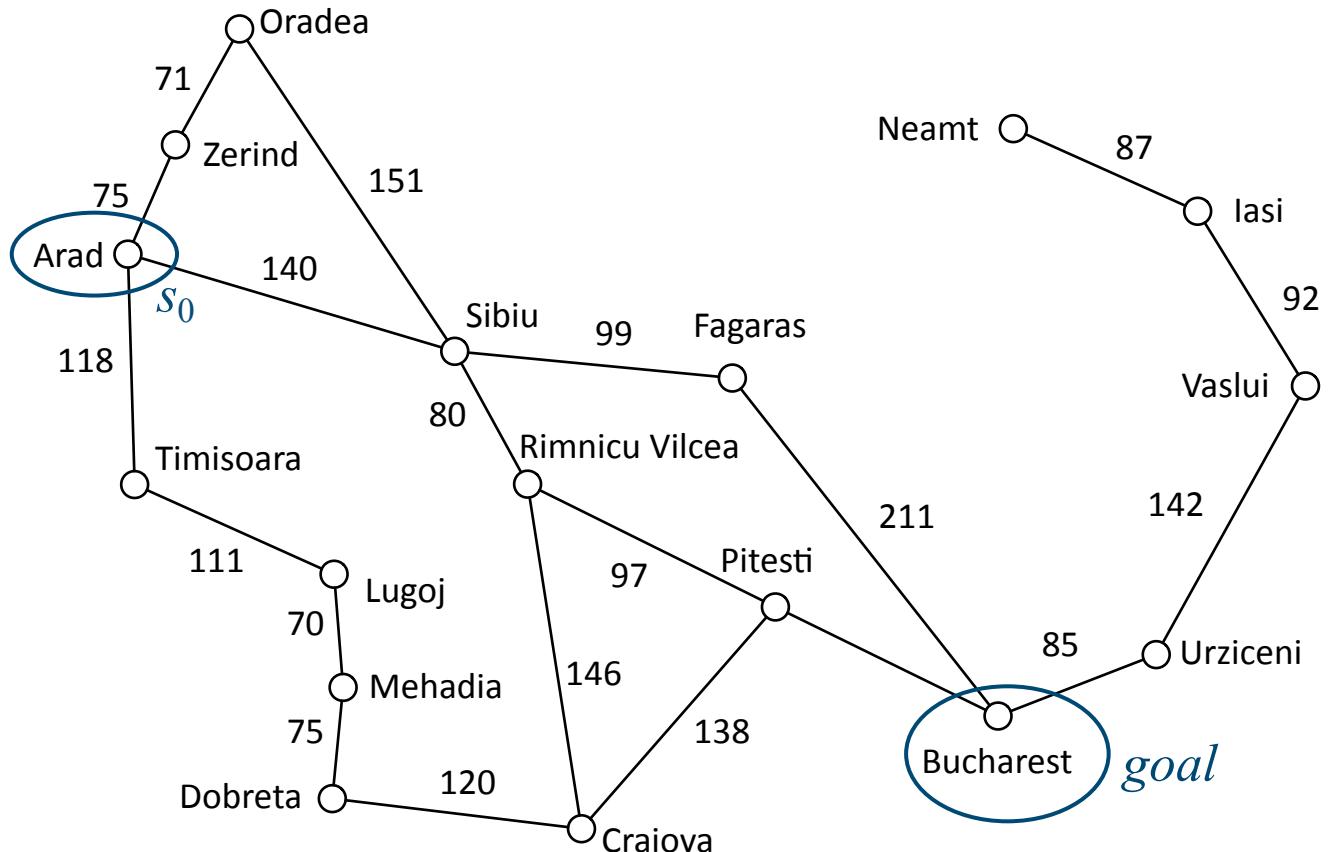
- Let  $h_1, h_2$  be admissible heuristic functions
- $h_2$  dominates  $h_1$  if  $\forall s, h_1(s) \leq h_2(s) \leq h^*(s)$

- 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  $h_1$

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

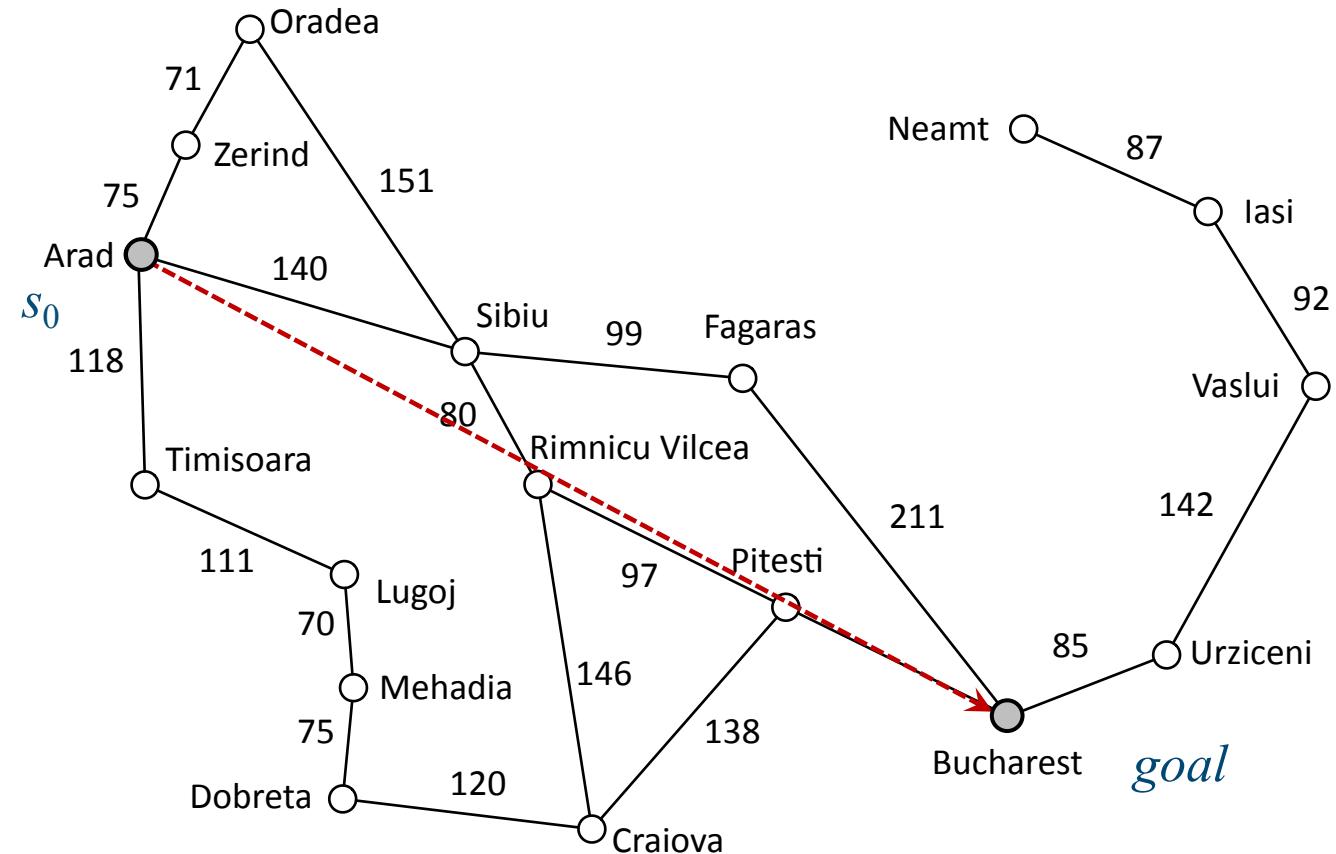
- A. Yes    B. No    C. Not sure



| straight-line dist.<br>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                                   |

# 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 2.3)



# 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  $h_1$  then (assuming A\* always resolves ties in favor of the same node)
  - ▶ 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
  - ▶  $\pi^*$  = best solution so far
  - ▶  $c^* = \text{cost}(\pi^*)$
  - ▶ prune  $v$  if  $f(v) \geq c^*$
  - ▶ when frontier is empty, return  $\pi^*$
- Properties
  - ▶ Termination, completeness, optimality same as A\*
  - ▶ Usually less memory, more time than A\*
  - ▶ Worst-case is like DFS:  $O(bl)$  memory,  $O(b^l)$  time

Deterministic-Search( $\Sigma, s_0, g$ )

```
Frontier ← {⟨⟩,  $s_0$ }  
Expanded ←  $\emptyset$   
 $c^* \leftarrow \infty; \pi^* \leftarrow \text{failure}$   
while  $Frontier \neq \emptyset$  do  
    select a node  $v = (\pi, s) \in Frontier$  (i)  
    remove  $v$  from  $Frontier$  and add it to  $Expanded$   
    if  $s$  satisfies  $g$  then return  $\pi$   
    if  $s$  satisfies  $g$  and  $\text{cost}(\pi) < c^*$  then  
         $c^* \leftarrow \text{cost}(\pi); \pi^* \leftarrow \pi$   
    else if  $f(v) < c^*$  then  
        Children ←  
         $\{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies } \text{pre}(a)\}$   
        prune 0 or more nodes from  
        Children,  $Frontier$ ,  $Expanded$  (ii)  
 $Frontier \leftarrow Frontier \cup Children$   
return failure  $\pi^*$ 
```

**Poll:** Have you seen DFBB before?

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

- Can express as modified version of Deterministic-Search
- Node (step i) selection like DFS:
  - ▶ Select  $v = (\pi, s) \in Children$  that has largest  $\text{length}(\pi)$
  - ▶ Tie-breaking: smallest  $h(s)$
- Pruning (step ii)
  - ▶ Like DFS, do cycle-checking and prune what recursive depth-first search would discard
- Additional pruning during node expansion
  - ▶ If  $f(v) \geq c^*$  then discard  $v$

# Comparisons

- If  $h$  is admissible, both A\* and DFBB will return optimal solutions
  - ▶ Usually DFBB generates more nodes, but A\* takes more memory
  - ▶ DFBB does badly in highly connected graphs (many paths to each state)
    - Can have exponentially worse running time than A\* (generates nodes exponentially many times)
  - ▶ DFBB best in problems where  $S$  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)

$\text{IDS}(\Sigma, s_0, g)$

for  $k = 1$  to  $\infty$  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:

$a, b, c$

$a, b, c, d, e, f, g$

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

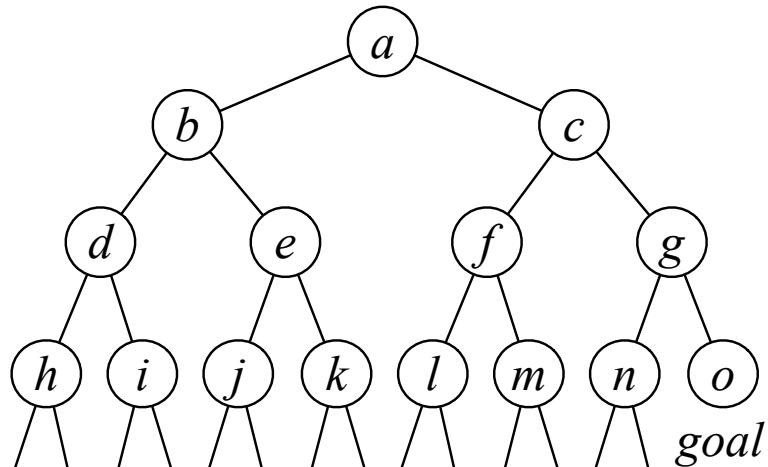
- Solution path  $\langle a, c, g, o \rangle$

- Total number of nodes generated:

$$3 + 7 + 15 = 25$$

- If goal is at depth  $d$  and branching factor is 2:

►  $\sum_{i=0}^d (2^{i+1} - 1) = \sum_{i=0}^d 2^{i+1} - \sum_{i=0}^d 1 = O(2^d)$



**Poll:** Have you seen Iterative Deepening before?

A. yes

B. no

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

**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)

$\text{IDS}(\Sigma, s_0, g)$

for  $k = 1$  to  $\infty$  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:

$a, b, c$

$a, b, c, d, e, f, g$

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

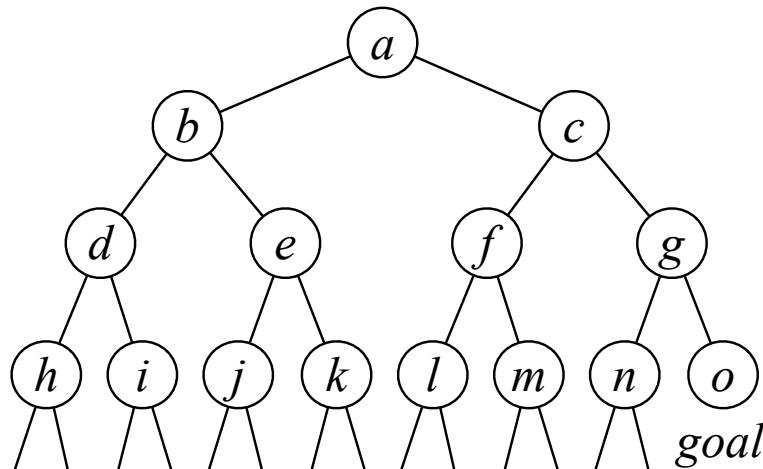
- Solution path  $\langle a, c, g, o \rangle$

- Total number of nodes generated:

$$3 + 7 + 15 = 25$$

- If goal is at depth  $d$  and branching factor is 2:

$$\triangleright \sum_{i=0}^d (2^{i+1} - 1) = \sum_{i=0}^d 2^{i+1} - \sum_{i=0}^d 1 = O(2^d)$$



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  $l$

# Summary

- 2.2 Forward State-Space Search
  - ▶ Forward-search, Deterministic-Search
  - ▶ cycle-checking
  - ▶ Breadth-first, depth-first, uniform-cost search
  - ▶ A\*, GBFS
  - ▶ DFBB, IDS

# Outline

- Chapter 2, part *a* (chap2a.pdf):
- 2.1 State-variable representation
  - Comparison with PDDL
  - 2.2 Forward state-space search
- Next →* 2.6 Incorporating planning into an actor
- 

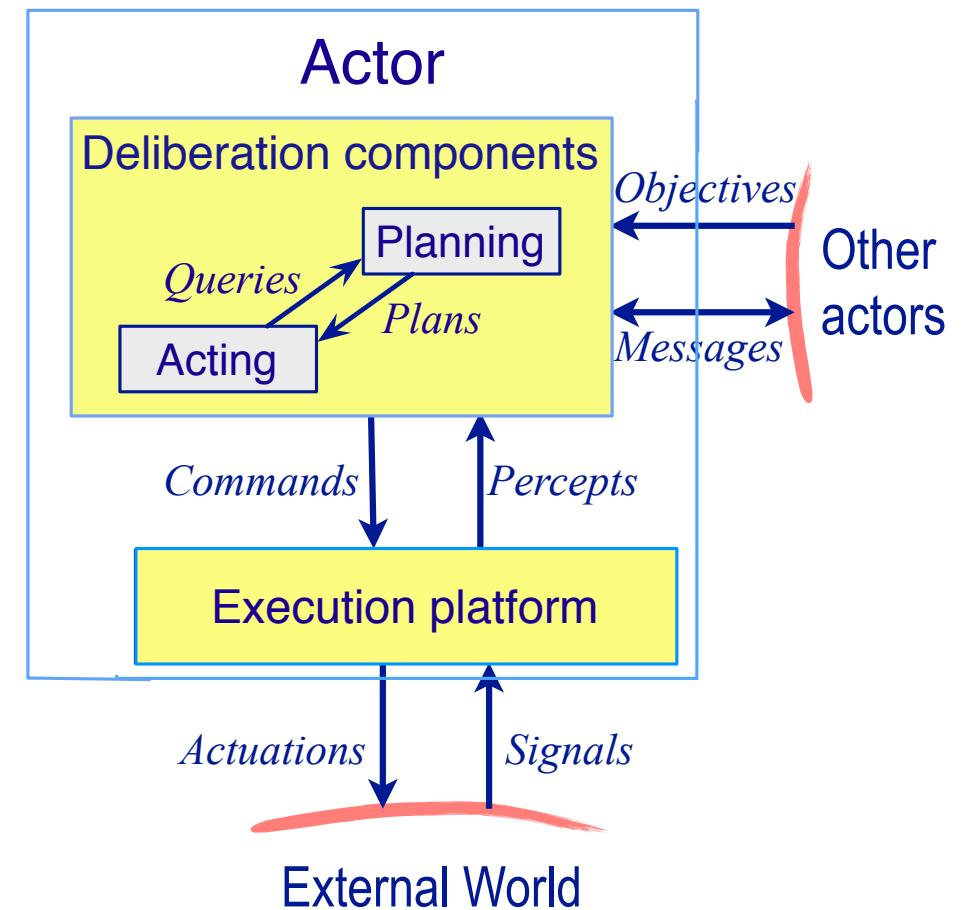
- Chapter 2, part *b* (chap2b.pdf):
- 2.3 Heuristic functions
  - 2.7.7 HTN planning
- 

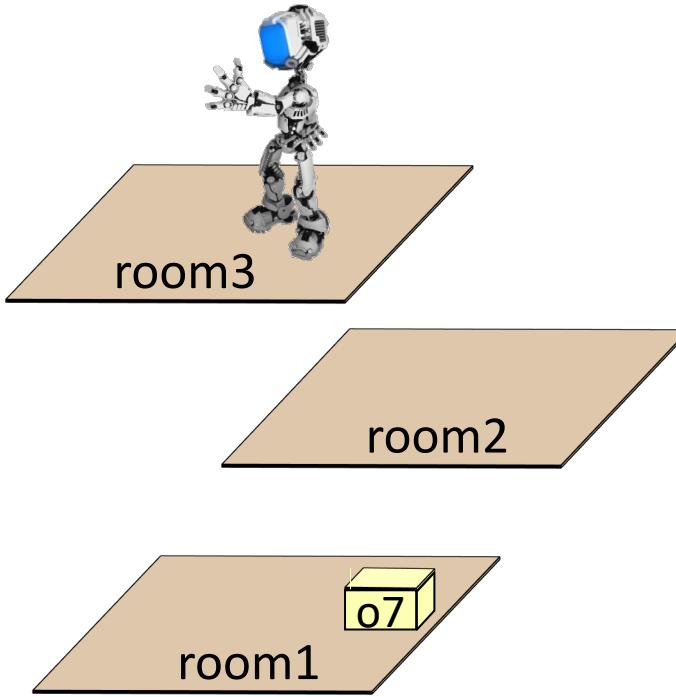
- Chapter 2, part *c* (chap2c.pdf):
- 2.4 Backward search
  - 2.5 Plan-space search
- 

- Additional slides:
- 2.7.8 LTL\_planning.pdf

## 2.6 Incorporating Planning into an Actor

- For classical planning we assumed
  - Finite, static world, just one actor
  - No concurrent actions, no explicit time
  - Determinism, no uncertainty
  - ▶ Sequence of states and actions  $\langle s_0, a_1, s_1, a_2, s_2, \dots \rangle$
- Most real-world environments don't satisfy the assumptions  
⇒ Errors in prediction
- OK if
  - ▶ errors occur infrequently, and
  - ▶ they don't have severe consequences
- What to do if an error *does* occur?





$a_1 = \text{go(r1, room3, hall)}$   
 $a_2 = \text{navigate(r1, hall, room1)}$   
 $a_3 = \text{take(r1, room1, o7)}$   
 $a_4 = \text{navigate(r1, room1, room2)}$   
 $a_5 = \text{put(r1, room2, o7)}$

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

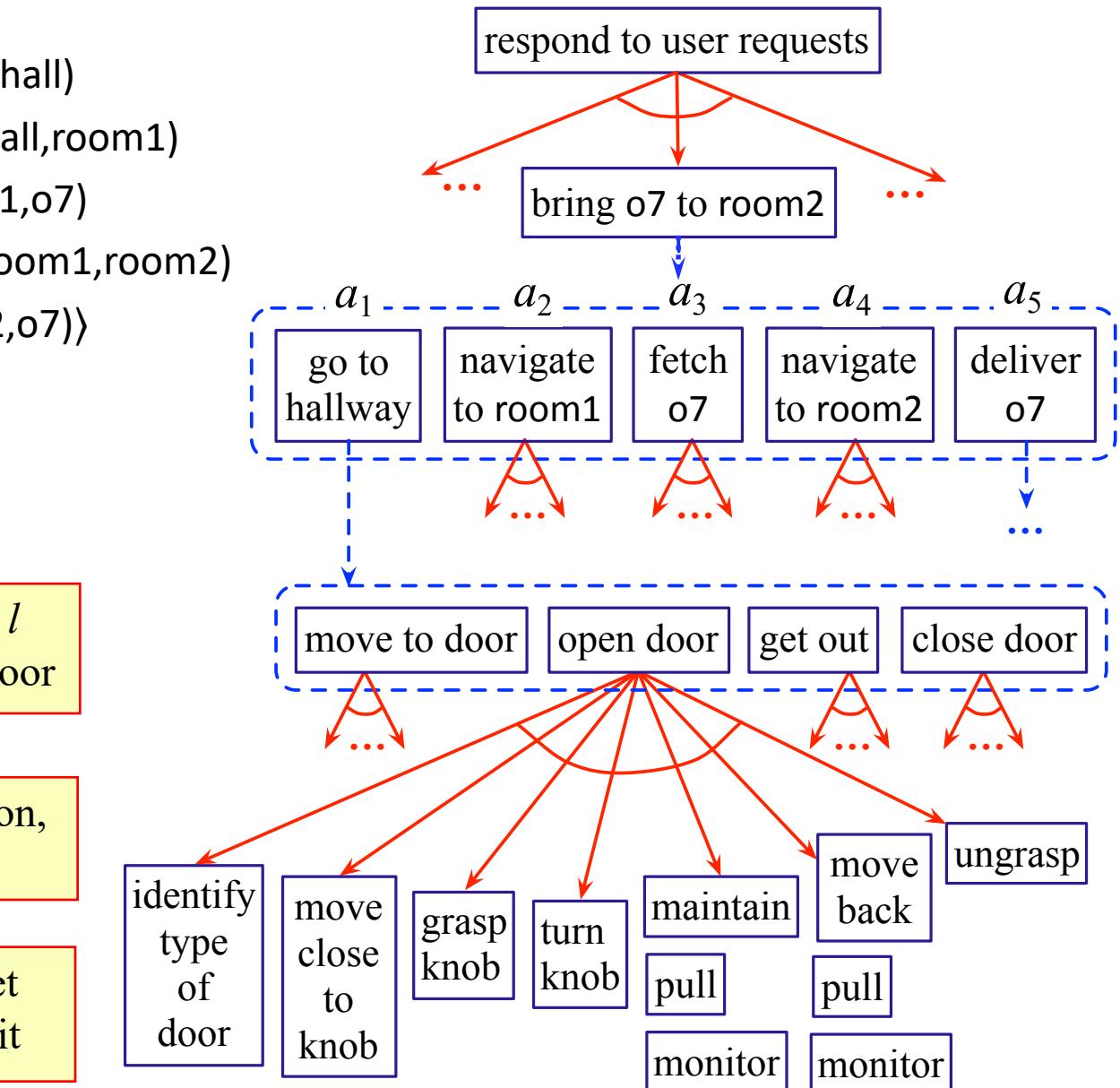
**navigate( $r, l, m$ )**  
 pre:  $\neg$ adjacent( $l, m$ ), loc( $r$ )= $l$   
 eff: loc( $r$ )  $\leftarrow m$

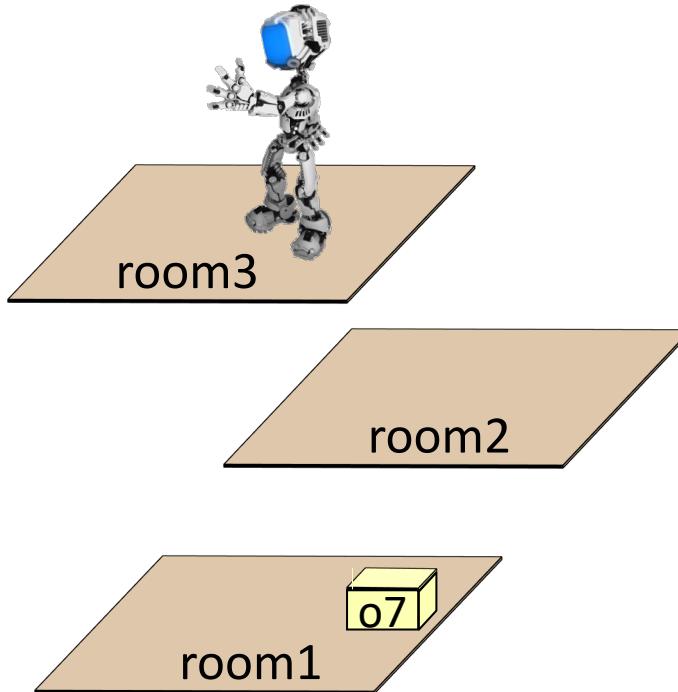
**take( $r, l, o$ )**  
 pre: loc( $r$ )= $l$ , loc( $o$ )= $l$ , cargo( $r$ )=nil  
 eff: loc( $o$ )  $\leftarrow r$ , cargo( $r$ )  $\leftarrow o$

ignores how to get from  $l$  to  $m$ , e.g., opening the door

ignores how do navigation, localization

ignores how to find  $o$ , get access to it, grasp it, lift it





$go(r,l,m)$

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

$navigate(r;l,m)$

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

$take(r;l,o)$

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

$a_1 = go(r1, room3, hall)$   
 $a_2 = navigate(r1, hall, room1)$   
 $a_3 = take(r1, room1, o7)$   
 $a_4 = navigate(r1, room1, room2)$   
 $a_5 = put(r1, room2, o7)$

ignores how to get from  $l$  to  $m$ , e.g., opening the door

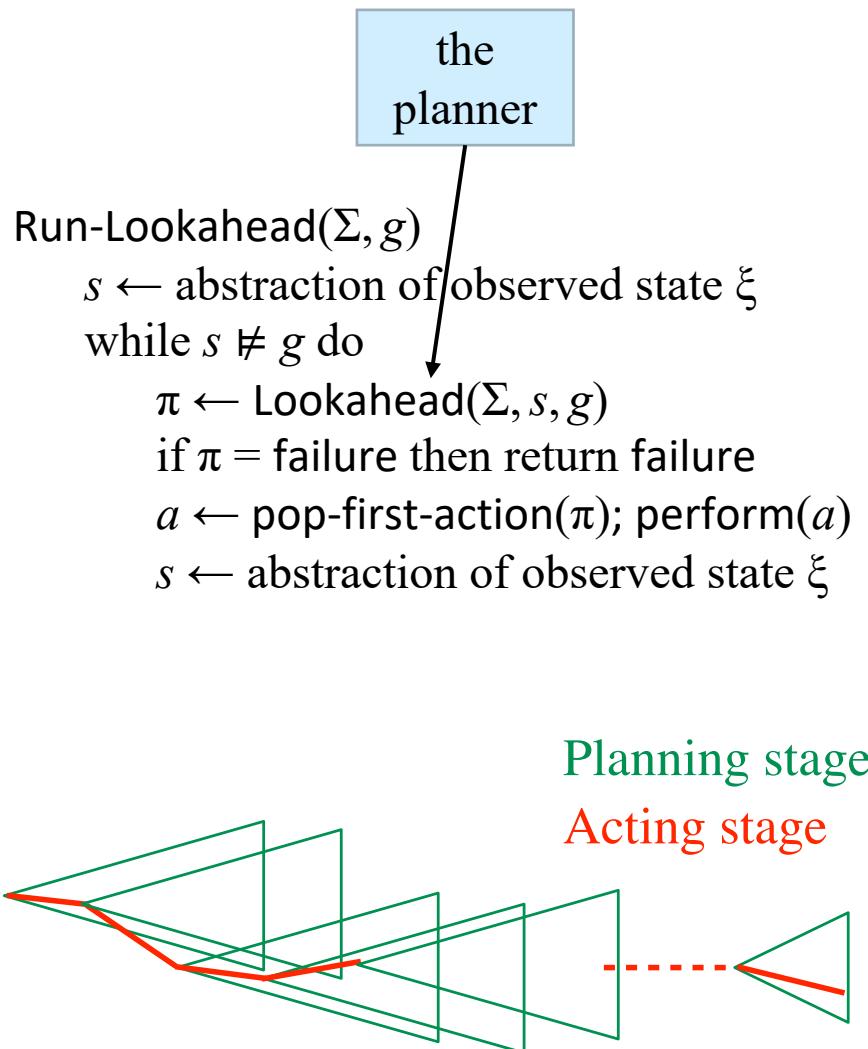
ignores how do navigation, localization

ignores how to find  $o$ , get access to it, grasp it, lift it

# Service Robot

- Some things that can go wrong:
  - *Execution failures*
    - locked door
    - robot battery goes dead
  - *Unexpected events*
    - class ends, hallway gets crowded
    - hallway closed for maintenance
  - *Incorrect information*
    - navigation error, go to wrong place
  - *Missing information*
    - where is  $o7$ ?
- How to detect and recover from errors?

# Using Planning in Acting



- Call Lookahead, obtain  $\pi$ , perform 1<sup>st</sup> action, call Lookahead again ...
- Useful when unpredictable things are likely to happen
  - ▶ Replans immediately
- Also useful with *receding horizon* search (e.g., as in chess programs):
  - ▶ Lookahead looks a limited distance ahead
- Potential problem:
  - ▶ Lookahead needs to return quickly
  - ▶ Otherwise, may pause repeatedly while waiting for Lookahead to return
  - ▶ What if  $\xi$  changes during the wait?

# Using Planning in Acting

Run-Lazy-Lookahead( $\Sigma, g$ )

$s \leftarrow$  abstraction of observed state  $\xi$

until  $s$  satisfies  $g$  do

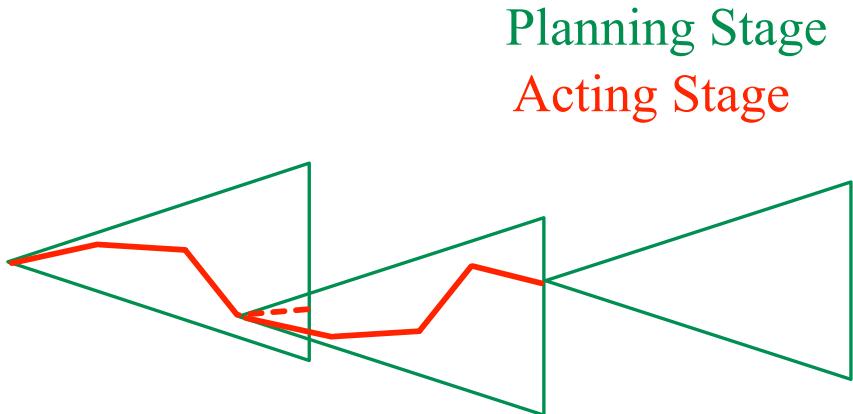
$\pi \leftarrow$  Lookahead( $\Sigma, s, g$ )

if  $\pi = \text{failure}$  then return failure

until  $\pi = \langle \rangle$  or  $s \models g$  or Simulate( $\Sigma, s, g, \pi$ ) = failure do

$a \leftarrow$  pop-first-action( $\pi$ ); perform( $a$ )

$s \leftarrow$  abstraction of observed state  $\xi$

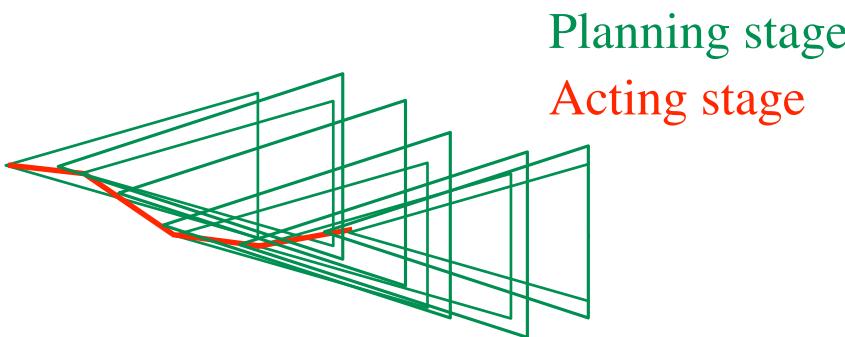


- Call Lookahead, execute the plan as far as possible, don't call Lookahead again unless necessary
- Simulate tests whether the plan will execute correctly
  - ▶ Lower-level refinement, physics-based simulation
- What if you don't have a simulation program?
  - ▶ Could write Simulate(...) to test whether  $\gamma(s, \pi) \models g$ 
    - or test whether  $s = \gamma(s', a)$ , where  $s'$  is the previous state
- Potential problems
  - ▶ Simulate needs to return quickly
    - otherwise, may pause repeatedly,  $\xi$  may change
  - ▶ May might miss opportunities to replace  $\pi$  with a better plan

# Using Planning in Acting

## Run-Concurrent-Lookahead (basic idea)

- ▶ global  $s, \pi$
- ▶ thread 1:
  - loop:
    - ▶  $s \leftarrow$  observed state
    - ▶  $\pi \leftarrow \text{Lookahead}(\Sigma, s, g)$
- ▶ thread 2:
  - loop:
    - ▶  $a \leftarrow \text{pop-first-element}(\pi)$
    - ▶ perform  $a$
    - ▶ return if observed state  $\models g$



- Motivation: plan and act in a dynamically changing environment
  - ▶ Want a recent plan, rather than the old one that Run-Lazy-Lookahead would use
  - ▶ Want to get it quickly, rather than waiting like Run-Lookahead
- But there are several problems with the pseudocode
  - ▶ It ignores some implementation details
    - how to do locking
    - whether each thread has correct values for  $\pi$  and  $s$
  - ▶ If thread 2 performs any actions while Lookahead is running, we probably should restart Lookahead
    - Otherwise Lookahead will return a plan that's out-of-date
  - ▶ Another possibility:
    - If thread 2 is going to perform action  $a$ , have thread 1 run  $\text{Lookahead}(\Sigma, \gamma(s, a), g)$

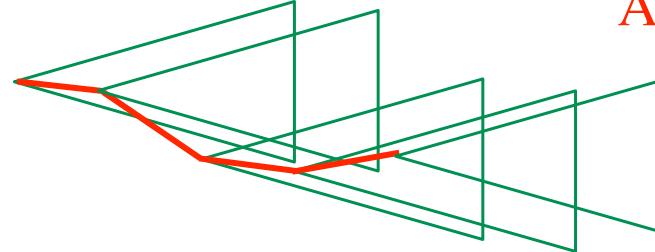
# How to do Lookahead

Some possibilities (can also combine these)

- **Full planning** (if the planner can solve the planning problem quickly enough)
- **Receding horizon**

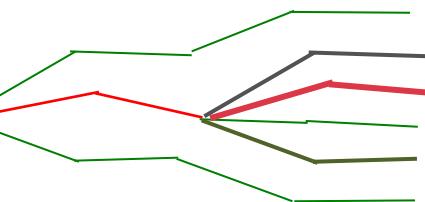
- ▶ Modify Lookahead to search just part of the way to  $g$  (see next page)
- ▶ E.g., cut off search when one of the following exceeds a maximum threshold:
  - plan length, plan cost, computation time

Planning stage  
Acting stage



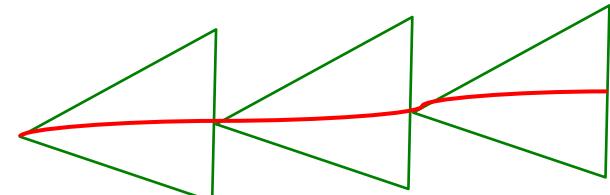
- **Sampling**

- ▶ Modify Lookahead to do a *Monte Carlo rollout*
  - Depth-first search with random node selection and no backtracking
- ▶ Call Lookahead several times, choose the plan that looks best
- ▶ Best-known example of this: the UCT algorithm (see Chapter 6)



- **Subgoaling**

- ▶ Tell Lookahead to plan for some subgoal  $g_1$ , rather than  $g$  itself
- ▶ Once the actor has achieved  $g_1$ , tell Lookahead to plan for the next subgoal  $g_2$
- ▶ And so forth until the actor reaches  $g$



# Receding-Horizon Search

Deterministic-Search( $\Sigma, s_0, g$ )

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

$Expanded \leftarrow \emptyset$

while  $Frontier \neq \emptyset$  do

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

    remove  $v$  from  $Frontier$

    add  $v$  to  $Expanded$

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

$Children \leftarrow \{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies } \text{pre}(a)\}$

    prune 0 or more nodes from

$Children, Frontier, Expanded$  (ii)

$Frontier \leftarrow Frontier \cup Children$

return failure

- Lookahead = modified version of Deterministic-Search

- Before line (i), put something like one of these:

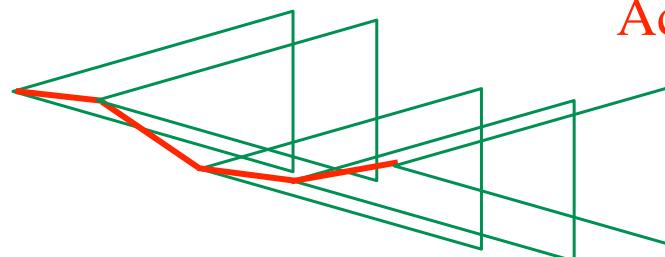
- *time-based cutoff*: if  $\text{time-left}() = 0$  then return  $\pi$
    - *length-based cutoff*: if  $|\pi| > l_{\max}$  then return  $\pi$
    - *cost-based cutoff*: if  $f(v) > c_{\max}$  then return  $\pi$
    - *closeness to goal*: if  $h(s) \leq \varepsilon$  then return  $\pi$

- Length-based and cost-based make sense if you're doing GBFS or AI, but not if you're doing DFS

- Could modify DFBB to use  $\pi^* = \text{least costly partial solution of length } \leq l_{\max}$

Planning stage

Acting stage



# Subgoaling Example

- Killzone 2
  - ▶ “First-person shooter” game, ≈ 2009
  - ▶ widely acclaimed at the time
- Special-purpose AI planner
  - ▶ Plans enemy actions at the squad level
    - Subproblems; plans are maybe 4–6 actions long
  - ▶ Different planning algorithm from what we’ve discussed so far
    - ▶ HTN planning (see Section 2.7.7)
    - Quickly generates a plan for a subgoal
    - Replans several times per second as the world changes
- Why it worked:
  - ▶ Don’t *want* to get the best possible plan
  - ▶ Need actions that appear believable and consistent to human users
  - ▶ Need them very quickly



# Summary

- 2.6 Incorporating Planning into an actor
  - ▶ Things that can go wrong while acting
  - ▶ Algorithms
    - Run-Lookahead,
    - Run-Lazy-Lookahead,
    - Run-Concurrent-Lookahead
  - ▶ Lookahead
    - receding-horizon search
    - sampling
    - subgoaling

# 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