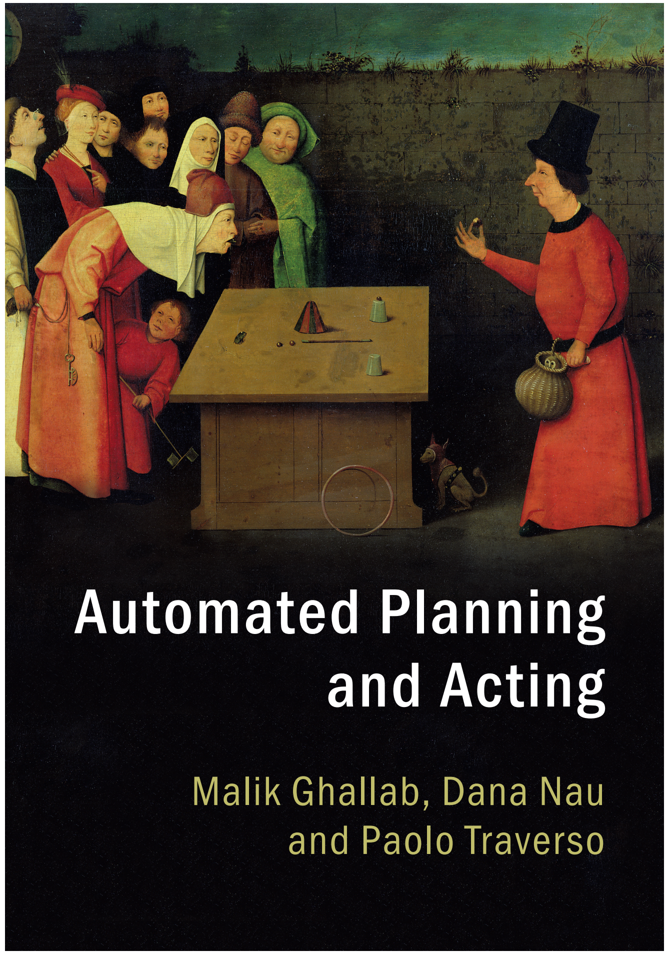
Last update: 3:25 PM, March 10, 2022

Chapter 2
Deliberation with Deterministic Models

Sections 2.1, 2.2, 2.6

Dana Nau
University of Maryland



http://www.laas.fr/planning

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 $\langle s_0, a_1, s_1, a_2, s_2, \ldots\rangle$
- 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

Domain Model

State-transition system or *classical planning domain*:

- $\Sigma = (S,A,\gamma,\text{cost})$ or (S,A,γ)
  - S - finite set of *states*
  - A - finite set of *actions*
  - $\gamma: S \times A \to S$

    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
      - $\text{Domain}(a) = \{s \in S \mid a \text{ is applicable in } s\}$
      - $\text{Range}(a) = \{\gamma(s,a) \mid s \in \text{Domain}(a)\}$
  - cost: $S \times A \to \mathbb{R}^+$ or cost: $A \to \mathbb{R}^+$
    - optional; default is $\text{cost}(a) \equiv 1$
    - money, time, something else

- *plan*:
  - a sequence of actions $\pi = \langle a_1, \ldots, a_n \rangle$
- π is *applicable* in s_0 if the actions are applicable in the order given

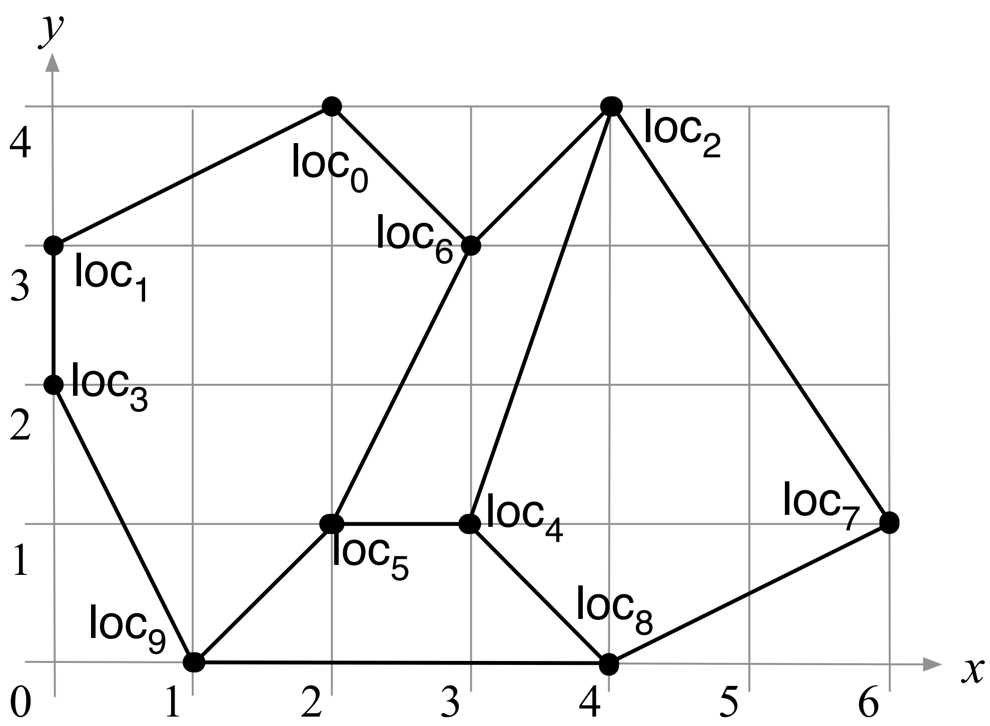
$$\gamma(s_0, a_1) = s_1$$
$$\gamma(s_1, a_2) = s_2$$
$$\ldots$$
$$\gamma(s_{n-1}, a_n) = s_n$$

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

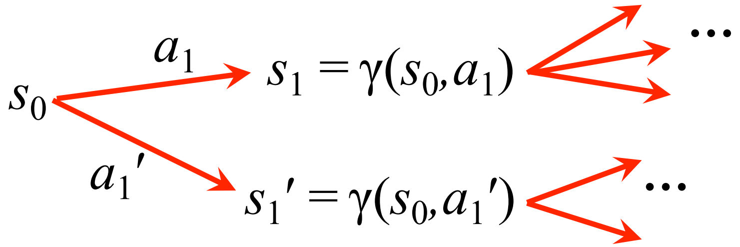
- *Classical planning problem*:
  - $P = (\Sigma, s_0, S_g)$
  - planning domain, initial state, set of goal states
- *Solution* for P:
  - a plan π such that that $\gamma(s_0,\pi) \in S_g$

Representing Σ

- 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

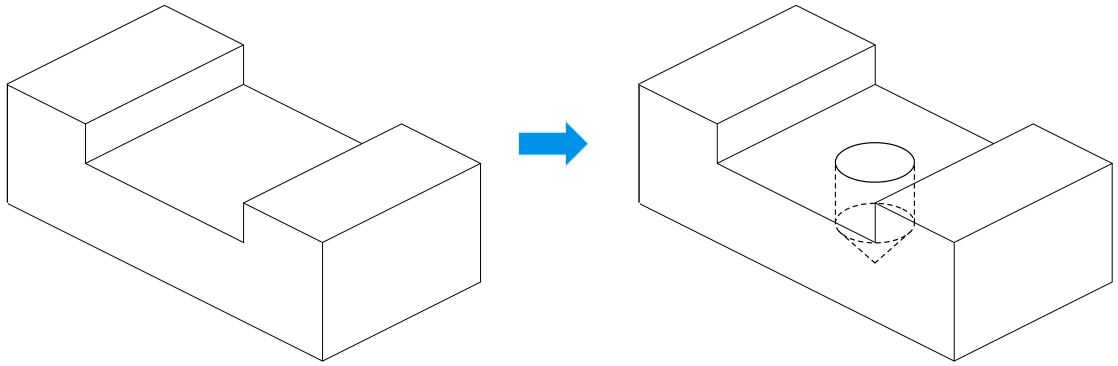


Domain-Specific Representation

- Tailor-made for a specific environment
- State: arbitrary data structure
- 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 ≡ 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



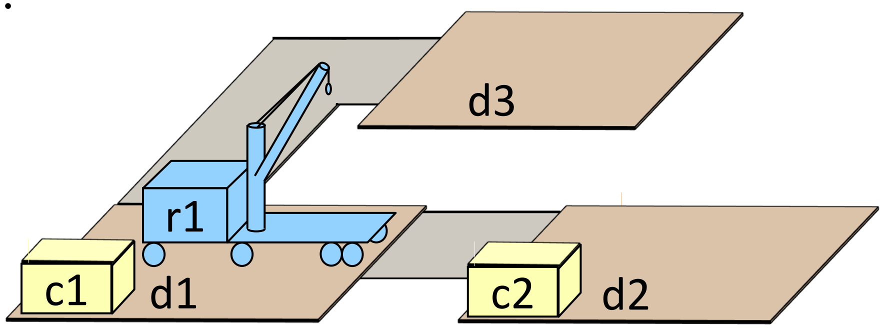
- Name: drill-hole
- Arguments:
  - ID codes for the machine and drill bit
  - annotated geometric model of the workpiece
  - description of the hole to be drilled
- Preconditions
  - *Capabilities*: can the machine and drill bit produce the desired hole?
  - *Current state*: Is the drill bit installed? 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
  - $B = Robots \cup Containers \cup Locs \cup \{\text{nil}\}$
    - $Robots = \{\text{r1}\}$
    - $Containers = \{\text{c1, c2}\}$
    - $Locs = \{\text{d1, d2, d3}\}$



- 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, …

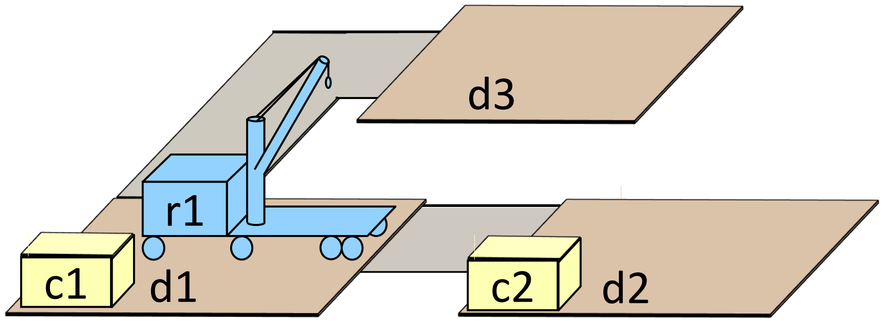
Rigid Properties

- Objects have two kinds of properties
  - *rigid* and *varying*
- *Rigid*: stays the same in every state
  - Can be described as a mathematical relation

    adjacent = {(d1,d2), (d2,d1), (d1,d3), (d3,d1)}
  - Or equivalently, a set of *ground atoms*

    adjacent(d1,d2), adjacent(d2,d1),

    adjacent(d1,d3), adjacent(d3,d1)
  - I'll use the two notations interchangeably



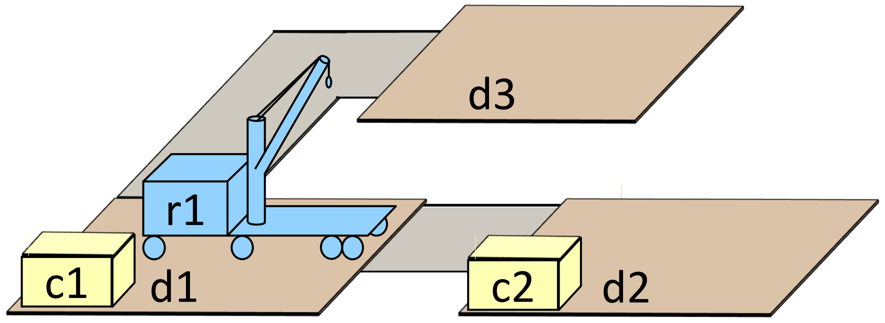
- Terminology from first-order logic:
  - *atom* ≡ *atomic formula* ≡ *positive literal* ≡ predicate symbol with list of arguments
    - *e.g.*, adjacent(*x*,d2)
  - an atom is *ground* (or *fully instantiated*) if it contains no variable symbols
    - *e.g.*, adjacent(d1,d2)
  - *negative literal* ≡ *negated atom* ≡ atom with a negation sign in front of it
    - *e.g.*, ¬ 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.*, loc(r1)
- Let X = {all state variables in the environment}

  e.g., X = {loc(r1), loc(c1), loc(c2), cargo(r1)}
- Each state variable $x \in X$ has a *range*

  = {all values that can be assigned to x}
    - Range(loc(r1)) = *Locs*
    - Range(loc(c1)) = Range(loc(c2)) = *Robots* ∪ *Locs*
    - Range(cargo(r1)) = *Containers* ∪ {nil}
- To abbreviate the "range" notation often I'll just say things like
    - loc(r1) ∈ *Locs*
    - loc(c1), loc(c2) ∈ *Robots* ∪ *Locs*

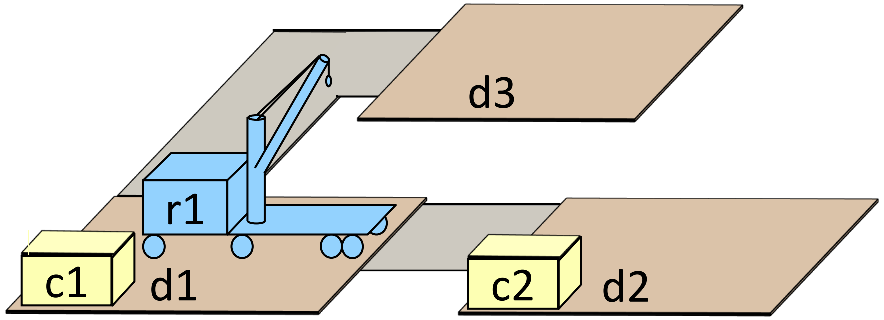


Instead of "domain", to avoid confusion with planning domains

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(loc(r1)) = d1, s_1(cargo(r1)) = nil,
s_1(loc(c1)) = d1, s_1(loc(c2)) = d2



- Mathematically, a function is a set of ordered pairs

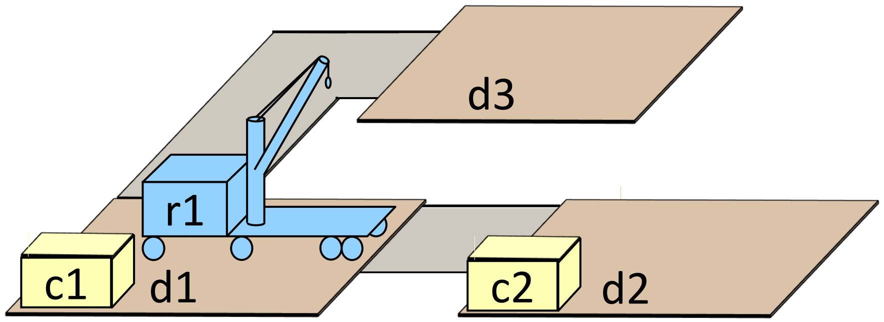
s_1 = {(loc(r1), d1), (cargo(r1), nil), (loc(c1), d1) , (loc(c2), d2)}

- Equivalently, write it as a set of *ground positive literals* (or *ground atoms*):

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

  - Here, we're using '=' as a predicate symbol

Action Templates



- Action *template* or *schema*: a parameterized set of actions

  α = (head, pre, eff, 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) ← m

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

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

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

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

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

Actions

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

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

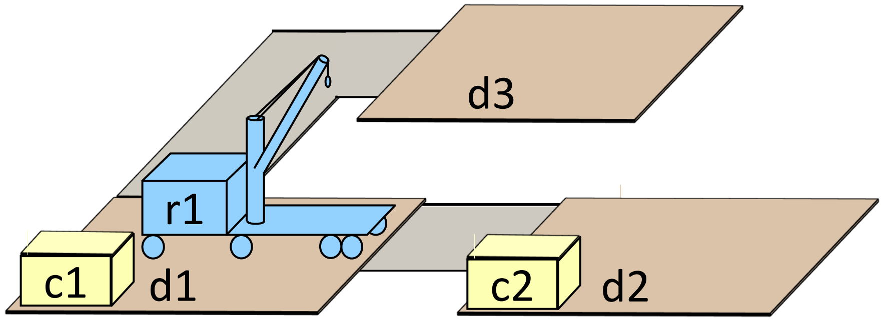
take(r,l,c)
  pre: cargo(r)=nil, loc(r)=l, loc(c)=l
  eff: cargo(r) ← c, loc(c) ← r

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

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

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

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



Actions

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

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

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

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

move(r1,d1,d2)
  pre: loc(r1)=d1, adjacent(d1,d2)
  eff: loc(r1) $\leftarrow$ d2

Poll. Let:
  $\mathcal{A}$ = {the action templates on this page}
  A = {all ground instances of members of $\mathcal{A}$}
How many move actions in A?

Answers:

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

Applicability

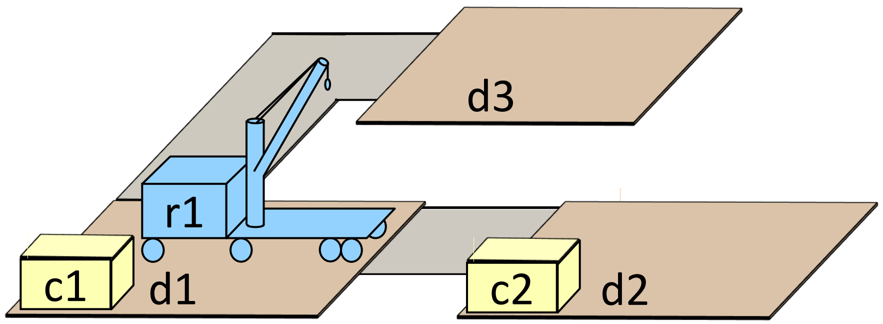
- a is *applicable* in s if
  - for every positive literal $l \in \text{pre}(a)$, $l \in s$ or l is in one of the rigid relations
  - for every negative literal $\neg l \in \text{pre}(a)$, $l \notin s$ and l isn't in any of the rigid relations

- Rigid relation

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

- *State*

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



- Action template

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

  Range(r) = *Robots*

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

- Applicable:

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

- Not applicable:

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

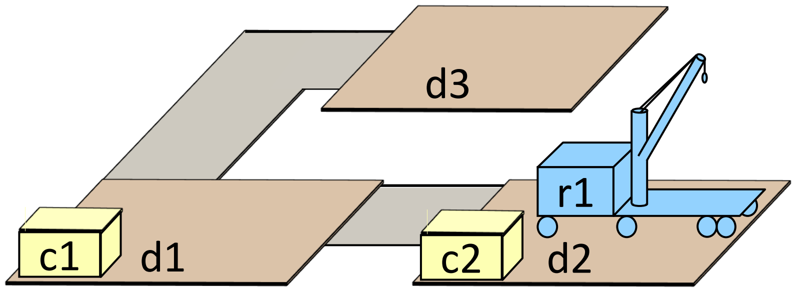
Poll: How many move actions are applicable in s_1?

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

State-Transition Function

- If a is applicable in s:
  - $\gamma(s,a)$ = {every literal in s that isn't changed in eff(a)}
    ∪ {every literal in eff(a)}

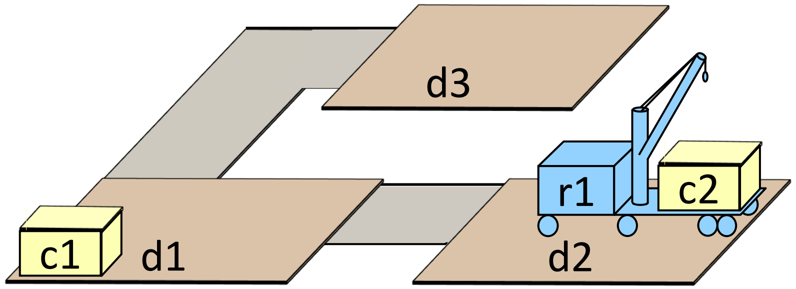
- s_2 = {loc(r1)=d2, cargo(r1)=nil, loc(c1)=d1, loc(c2)=d2}



- a = take(r1,d2,c2)
  pre: cargo(r1)=nil, loc(r1)=d2, loc(c2)=d2
  eff: cargo(r1) ← c2, loc(c2) ← r1

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

  {$\underbrace{\text{loc(r1)=d2, loc(c1)=d1,}}_{\text{from } s_2}$ $\underbrace{\text{cargo(r1)=c2, loc(c2)=r1}}_{\text{from eff}(a)}$}



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, 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, Range(y) $\subseteq B$

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

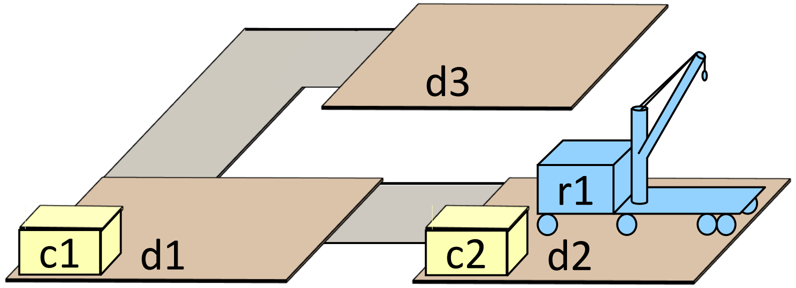
  $\gamma(s,a) = \{(x,w) \mid \mathrm{eff}(a) \text{ contains the effect } x \leftarrow w\}$
  $\cup \{(x,w) \in s \mid x \text{ isn't the target of any effect in } \mathrm{eff}(a)\}$

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

Interpretations

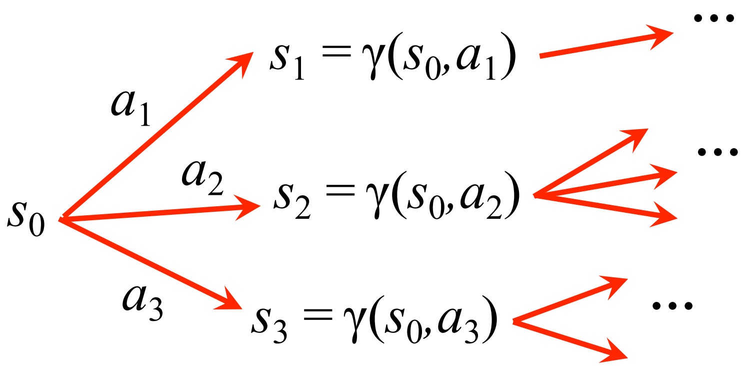
- Let s be a value-assignment function
  - s is a state only if the values make sense in the planning domain we're trying to represent
    - (relation to *model theory*)
- Can loc(c1)=r1 if cargo(r1)=nil?
  - Not in our intended *interpretation*
    - Mapping of symbols to what they represent

s_2 = {loc(r1)=d2,
cargo(r1)=nil,
loc(c1)=d1,
loc(c2)=d2}



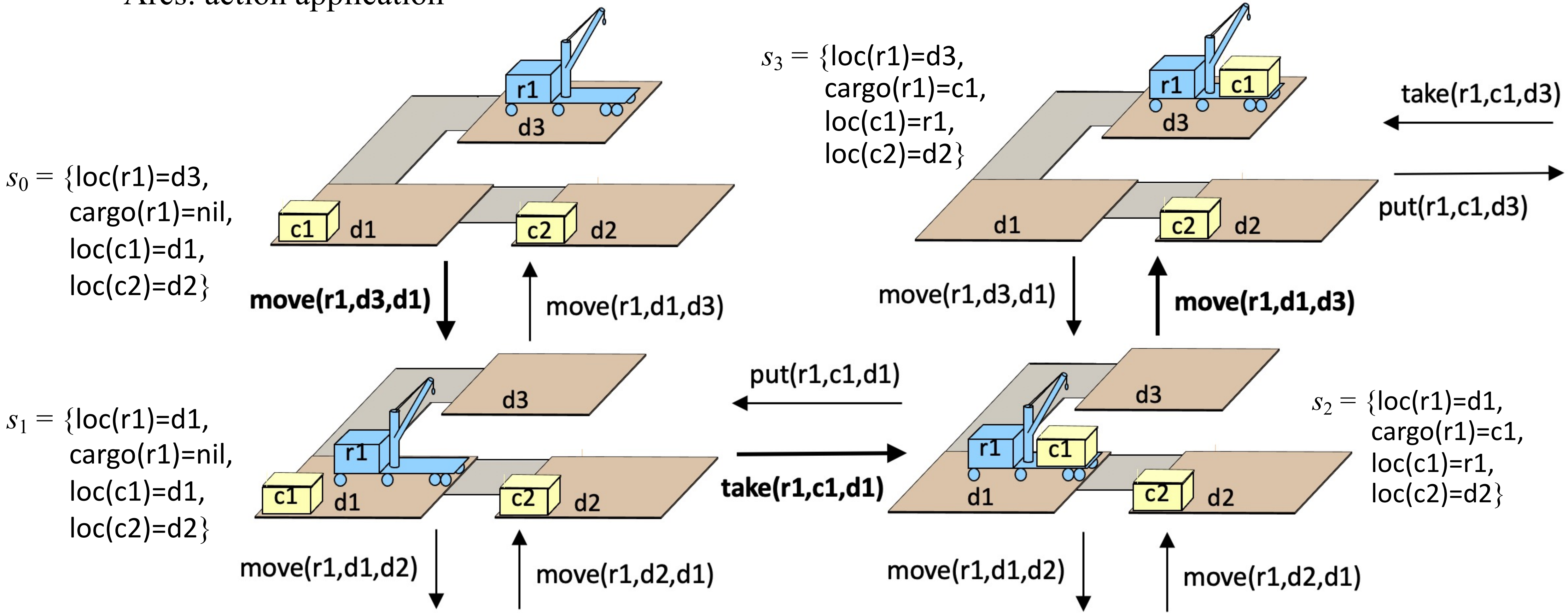
- Can both loc(c1)=r1 and loc(c2)=r1?
  - In our intended interpretation, can a robot carry more than one object at a time?

- 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



State Space

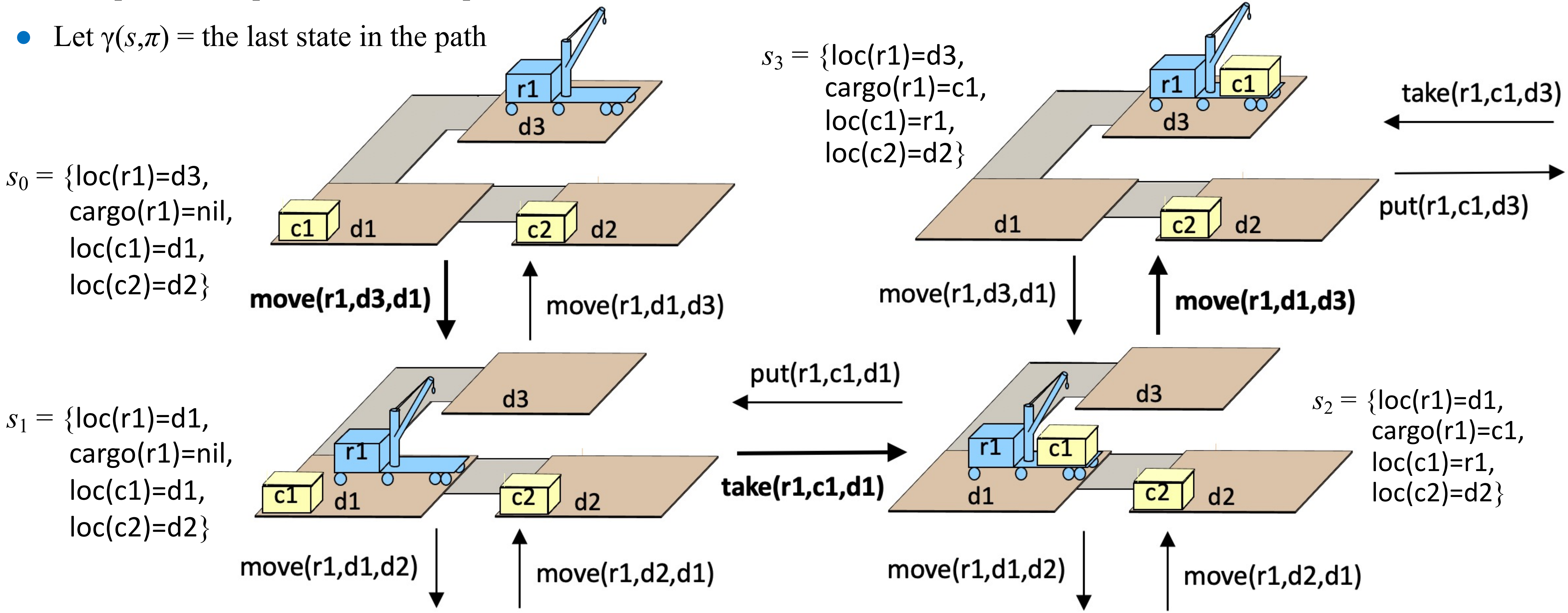
- *State Space*: a directed graph
  - Nodes = states of the world
  - Arcs: action application



Applying a Plan

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

- If π = ⟨move(r1,d3,d1), take(r1,d1,c1), move(r1,d1,d3)⟩ then $\gamma(s_0,\pi) = s_3$



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*

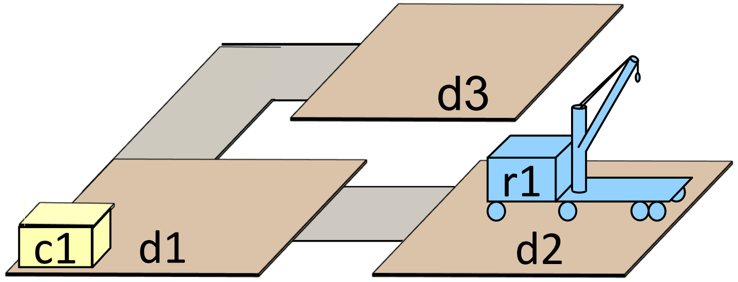
- S_g = {all states in S that satisfy g}
  = {$s \in S \mid s \cup R$ contains every positive literal in g, and none of the negative literals in g}

- If $\gamma(s_0, \pi)$ satisfies g (or equivalently, $\gamma(s_0, \pi) \in S_g$) then π is a *solution* for P

Poll: How many solutions of length 3?

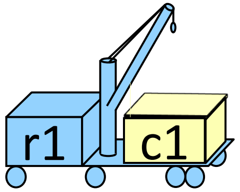
A. 1	B. 2	C. 3	D. 4	E. 5
F. 6	G. 7	H. 8	I. 9	J. other

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



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

g = {cargo(r1)=c1}



⟨move(r1,d2,d1), take(r1,d1,c1)⟩ is a solution of length 2

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) $\leftarrow l$

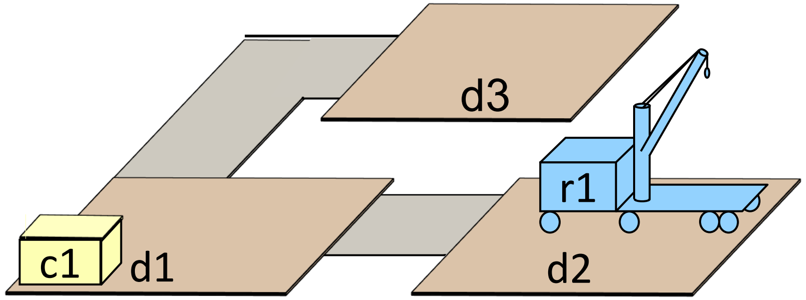
Range(r) = *Robots*
Range(l) = *Locs*
Range(m) = *Locs*

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

adjacent(l,m)		- location l is adjacent to m
loc$(r) = l$	$\longrightarrow$ loc(r,l)	- robot r is at location l
loc$(c) = r$	$\longrightarrow$ loc(c,r)	- container c is on robot r
cargo$(r) = c$	$\longrightarrow$ loaded(r)	- there's a container on r

why not loaded(r,c)?



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

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

Poll: Should s_0 also contain ¬ loaded(r1) ?

A: yes B: no

C: unsure

Classical planning operators

- Action templates

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

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

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

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

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

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

- Classical planning operators

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

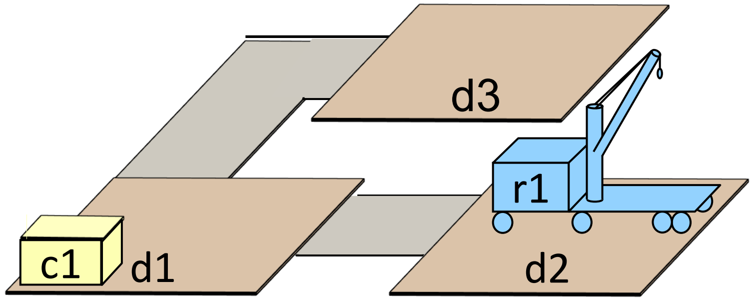
take(r,l,c)
  pre: ¬loaded(r), loc(r,l), loc(c,l)
  eff: loaded(r), ¬loc(c,l), loc(c,r)

put(r,l,c)
  pre: loc(r,l), loc(c,r)
  eff: ¬loaded(r), loc(c,l), ¬loc(c,r)

Poll: Does move really need to include ¬loc(r,l) ?

A: yes B: no

C: unsure



Actions

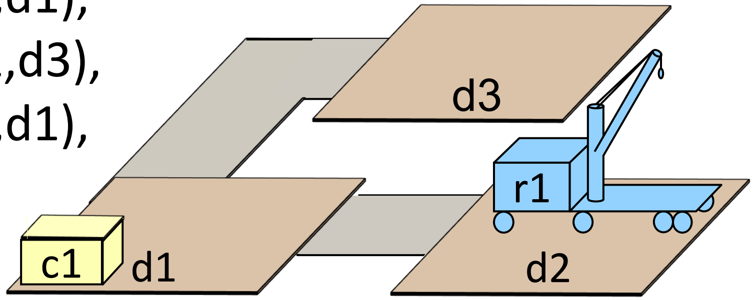
- Planning operator:

  o: move(r,l,m)
  pre: loc(r,l), adjacent(l,m)
  eff: ¬loc(r,l), loc(r,m)

- Action:

  a_1: move(r1,d2,d1)
  pre: loc(r1,d2), adjacent(d2,d1)
  eff: ¬loc(r1,d2), loc(r1,d1)

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



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

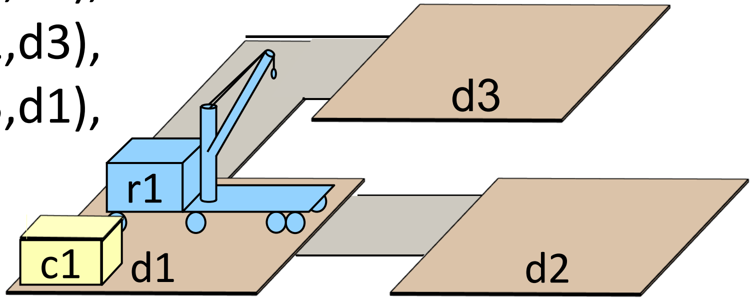
- a is applicable in state s iff

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

- If a is applicable in s then
  - $\gamma(s,a) = (s \setminus \text{eff}^-(a)) \cup \text{eff}^+(a)$

meaning?

$\gamma(s_0, a_1)$ = {adjacent(d1,d2),
adjacent(d2,d1),
adjacent(d1,d3),
adjacent(d3,d1),
loc(c1,d1),
loc(r1,d1)}



Discussion

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

State-variable rep.

Classical rep.

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

- 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

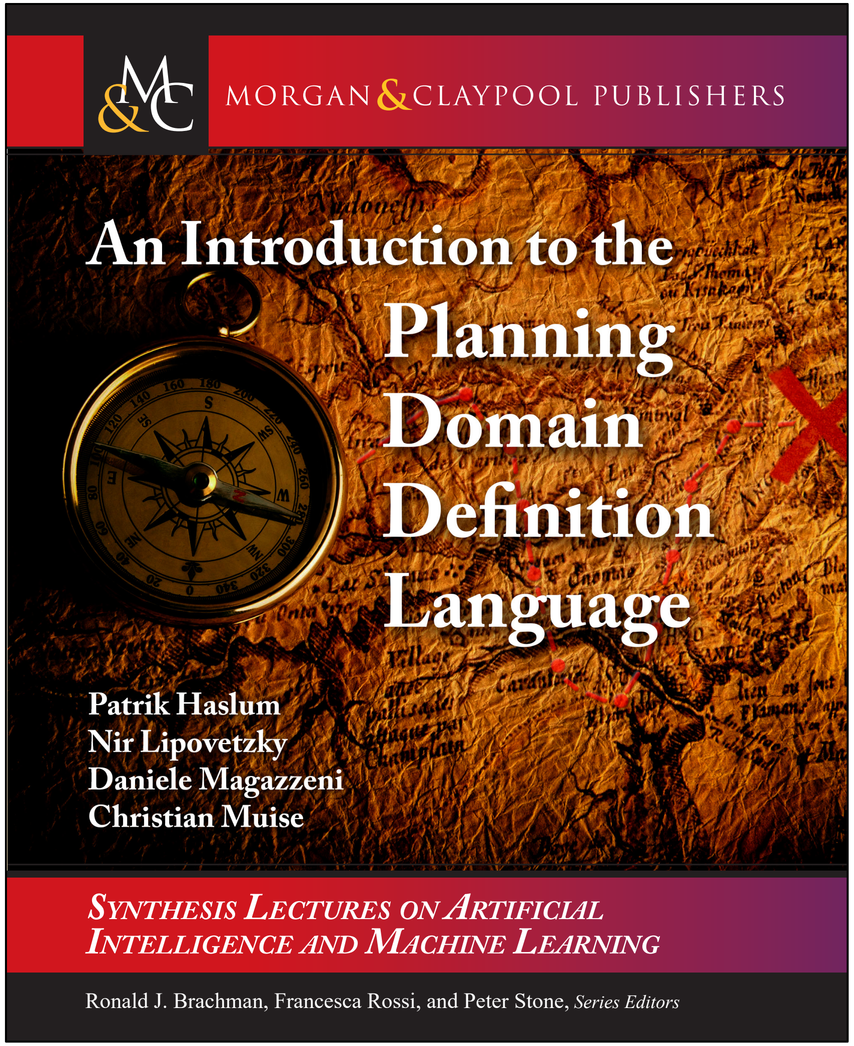
Poll: Could we instead use $x_P(b_1,\ldots,b_{k-1})=b_k$?

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



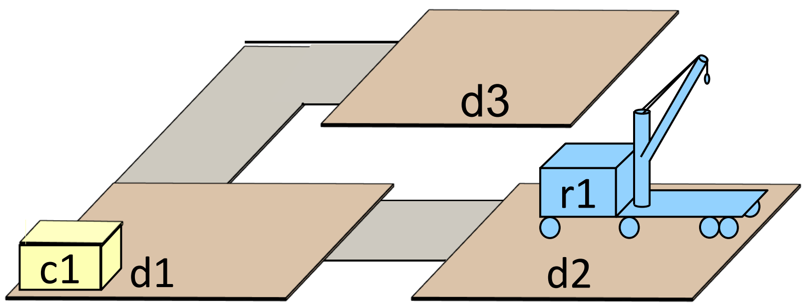
Example domain

- Classical representation:

move(r,l,m)
Precond: loc(r,l), adjacent(l,m)
Effects: ¬loc(r,l), loc(r,m)

take(r,l,c)
Precond: loc(r,l), loc(c,l), ¬loaded(r)
Effects: loc(c,r), ¬loc(c,l), loaded(r)

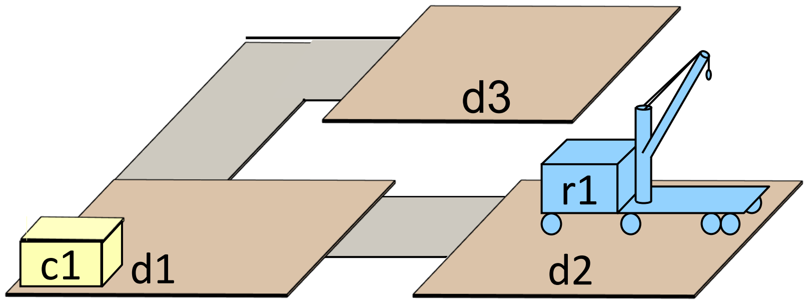
put(r,l,c)
Precond: loc(r,l), loc(c,r)
Effects: loc(c,l), ¬loc(c,r), ¬loaded(r)



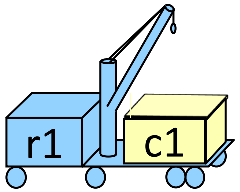

```
(define (domain example-domain-1)
    (requirements :negative-preconditions)

    (: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 = {adjacent(d1,d2), adjacent(d2,d1),
adjacent(d1,d3), adjacent(d3,d1),
loc(c1,d1), loc(r1,d2)}



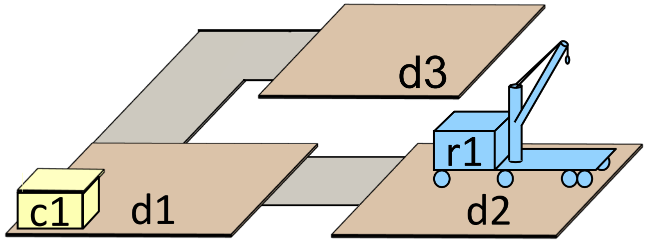
g = {loc(c1,r1)}

```
(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 = Movable_objects* ∪ *Locs*
  - *Movable_objects = Robots* ∪ *Containers*
  - *Robots* = {r1}
  - *Containers* = {c1}
  - *Locs* = {d1, d2, d3}



- Parameter ranges
  - *r* ∈ *Robots*
  - *l,m* ∈ *Locs*
  - *c* ∈ *Containers*

```
(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)
        (loloaded ?r - robot)
        (adjacent ?l ?m - location))
```

like saying

Locations, Movable_objects ⊆ *B*

Robots, Containers ⊆ *Movable_objects*

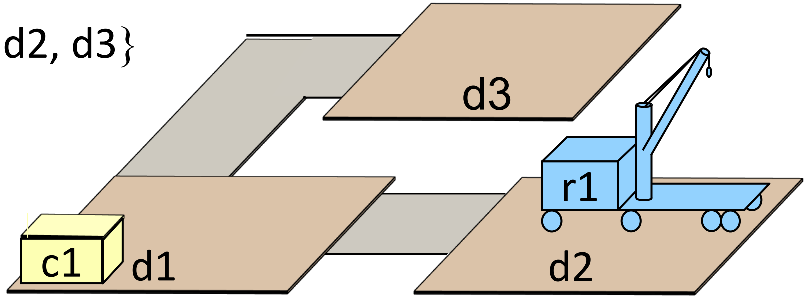
```
(:action move
 :parameters (?r - robot
              ?l ?m - location)
 :precondition (and (loc ?r ?l)
                    (adjacent ?l ?m))
 :effect (and (not (loc ?r ?l))
              (loc ?r ?m)))

(: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* ∈ *Robots*, *l* ∈ *Locs*, *c* ∈ *Containers*

Typed problem

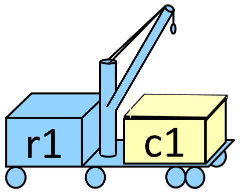
State-variable planning:

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



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

$g = \{$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

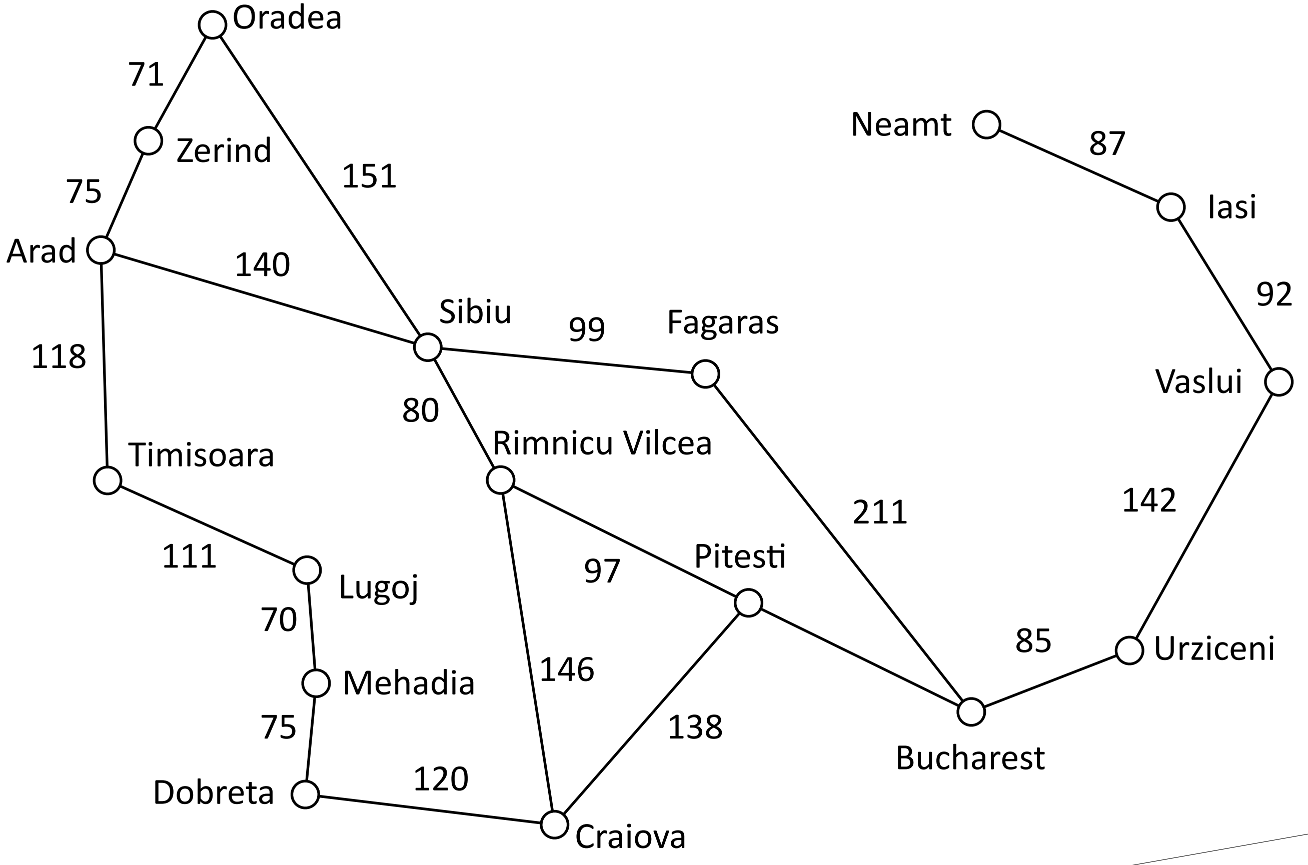
Section 2.1 of Ghallab *et al.* (2016)

- 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, γ
  - State-variable planning domains, plans, problems, solutions
  - Comparison with classical representation

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

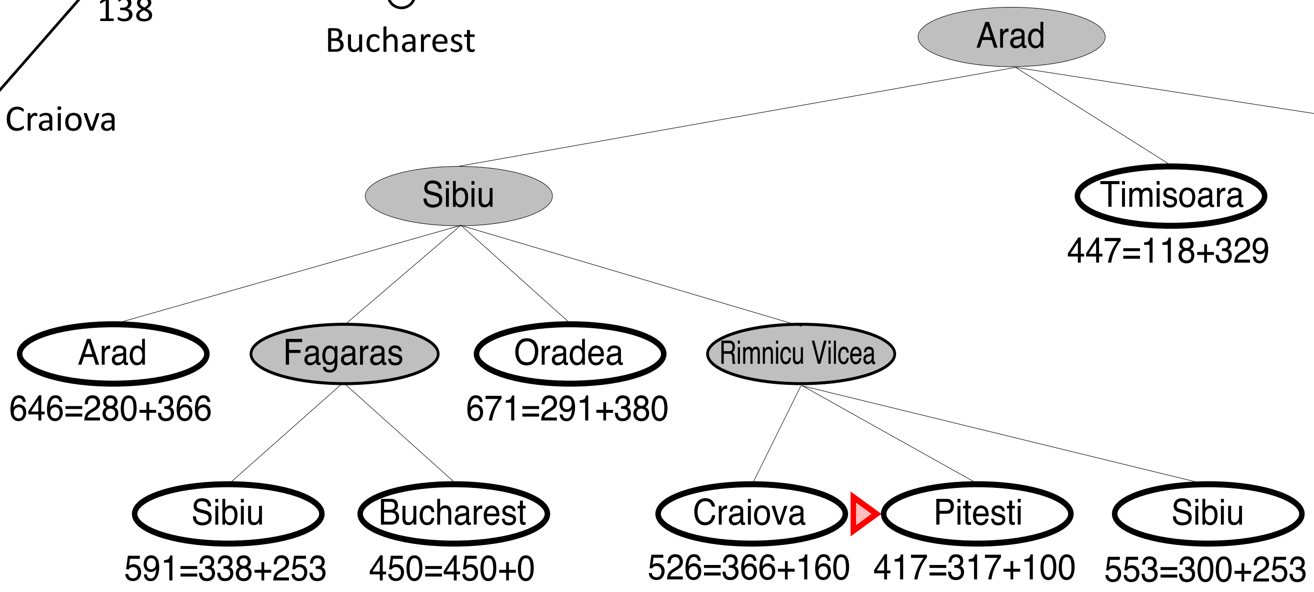
- Classical fragment of PDDL
  - Planning domains, planning problems
  - untyped, typed

Outline



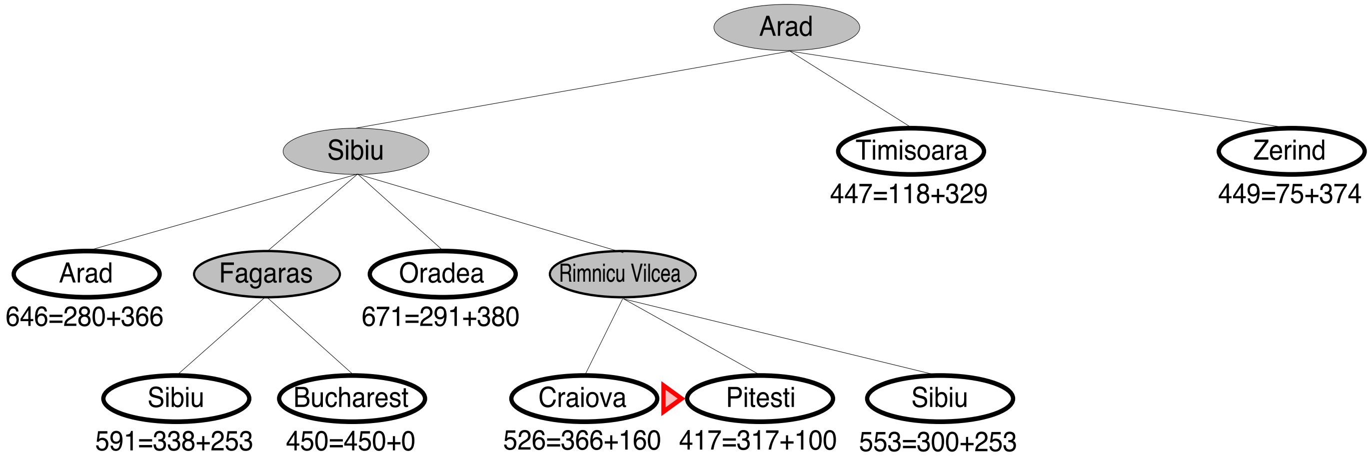
Planning as Search

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



Example: Russell & Norvig, *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.
- *ancestors* of ν
  = {nodes that have ν as a successor}
- *initial* or *starting* or *root* node $\nu_0 = (\langle\rangle, s_0)$
  - root of the search tree
- *path* in the search space: sequence of nodes $\langle \nu_0, \nu_1, \ldots, \nu_n \rangle$ 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

Forward Search

Forward-search (Σ, s_0, g)
    $s \leftarrow s_0$; $\pi \leftarrow \langle\rangle$
    loop
        if s satisfies g then return π
        $A' \leftarrow \{a \in A \mid a$ 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 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
  - 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 (Σ, s₀, g)
    s ← s₀;   π ← ⟨⟩
    loop
        if s satisfies g then return π
        A′ ← {a ∈ A | a is applicable in s}
        if A′ = ∅ then return failure
        nondeterministically choose a ∈ A′
        s ← γ(s,a);   π ← π.a
```

- 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

- Many of the algorithms in this class:

```
Deterministic-Search(Σ, s₀, g)
    Frontier ← {(⟨⟩, s₀)}
    Expanded ← ∅
    while Frontier ≠ ∅ do
        select a node ν = (π, s) ∈ Frontier          (i)
        remove ν from Frontier
        add ν to Expanded
        if s satisfies g then return π
        Children ←
                {(π.a, γ(s,a)) | s satisfies pre(a)}
        prune 0 or more nodes from
                Children, Frontier, Expanded         (ii)
        Frontier ← Frontier ∪ Children
    return failure
```

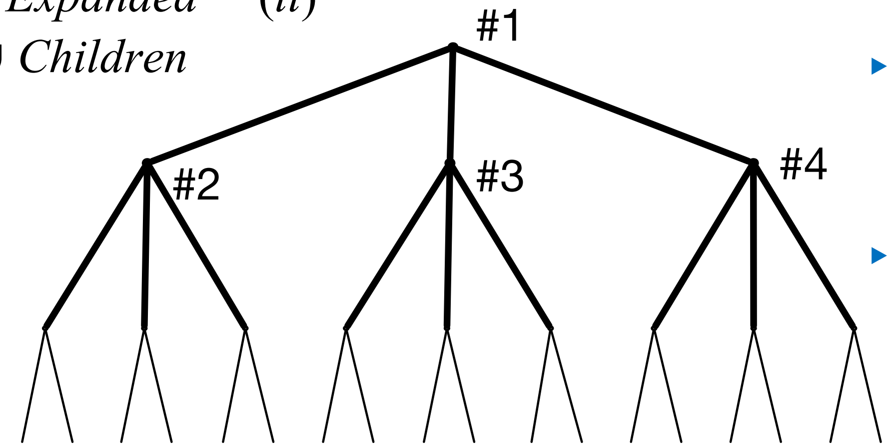
Deterministic Version

Deterministic-Search(Σ, s_0, g)
    Frontier $\leftarrow \{(\langle\rangle, s_0)\}$
    Expanded $\leftarrow \emptyset$
    while *Frontier* $\neq \emptyset$ do
        select a node $\nu = (\pi, s) \in$ *Frontier*   (*i*)
        remove ν from *Frontier*
        add ν to *Expanded*
        if s satisfies g then return π
        Children $\leftarrow$
            $\{(\pi.a, \gamma(s,a)) \mid s$ satisfies 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 (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)

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



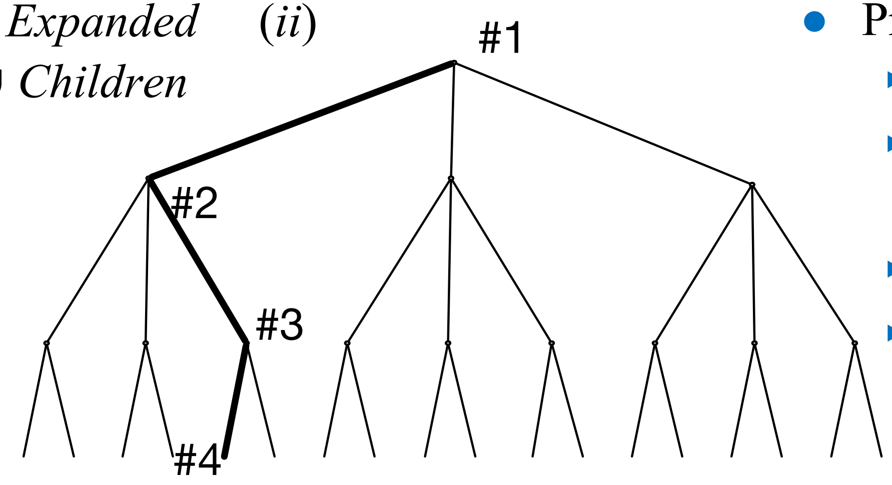
(*i*): Select (π,s) ∈ *Frontier* that has the smallest length(π), i.e., smallest number of edges
- Tie-breaking rule: select oldest

(*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 $O(b|S|)$
  - where
    - b = max branching factor
    - $|S|$ = number of states in S

Depth-First Search (DFS)

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

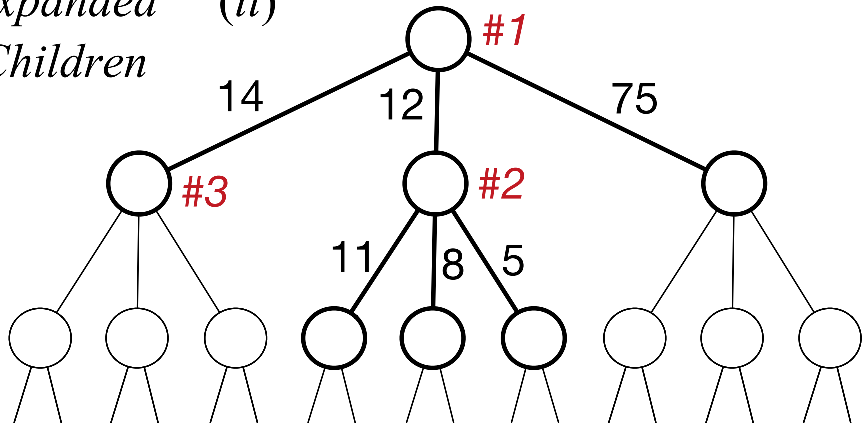
Uniform-Cost Search

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

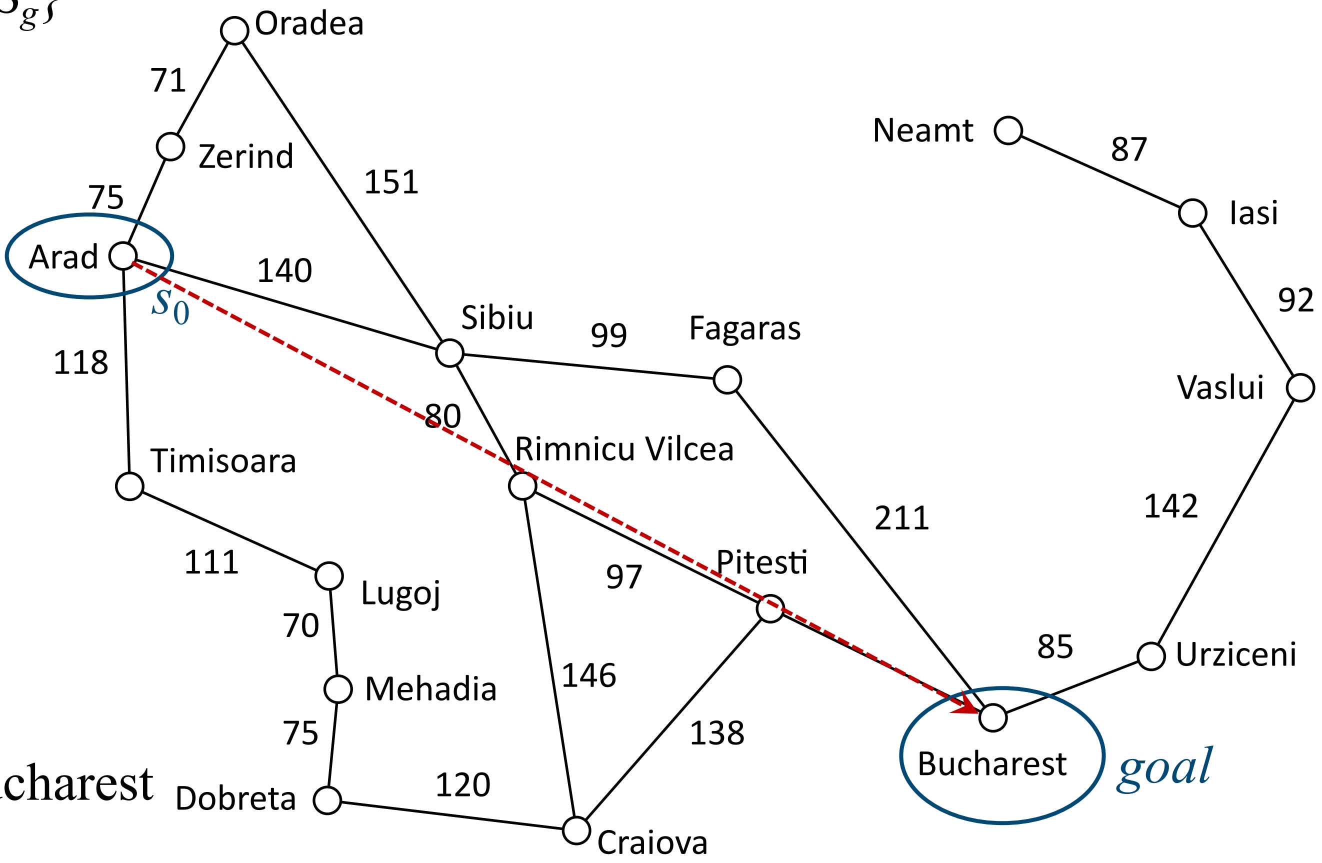


Poll: If node ν is expanded before node ν′, then how are cost(ν) and cost(ν′) related?

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

- 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



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

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

Greedy Best-First Search (GBFS)

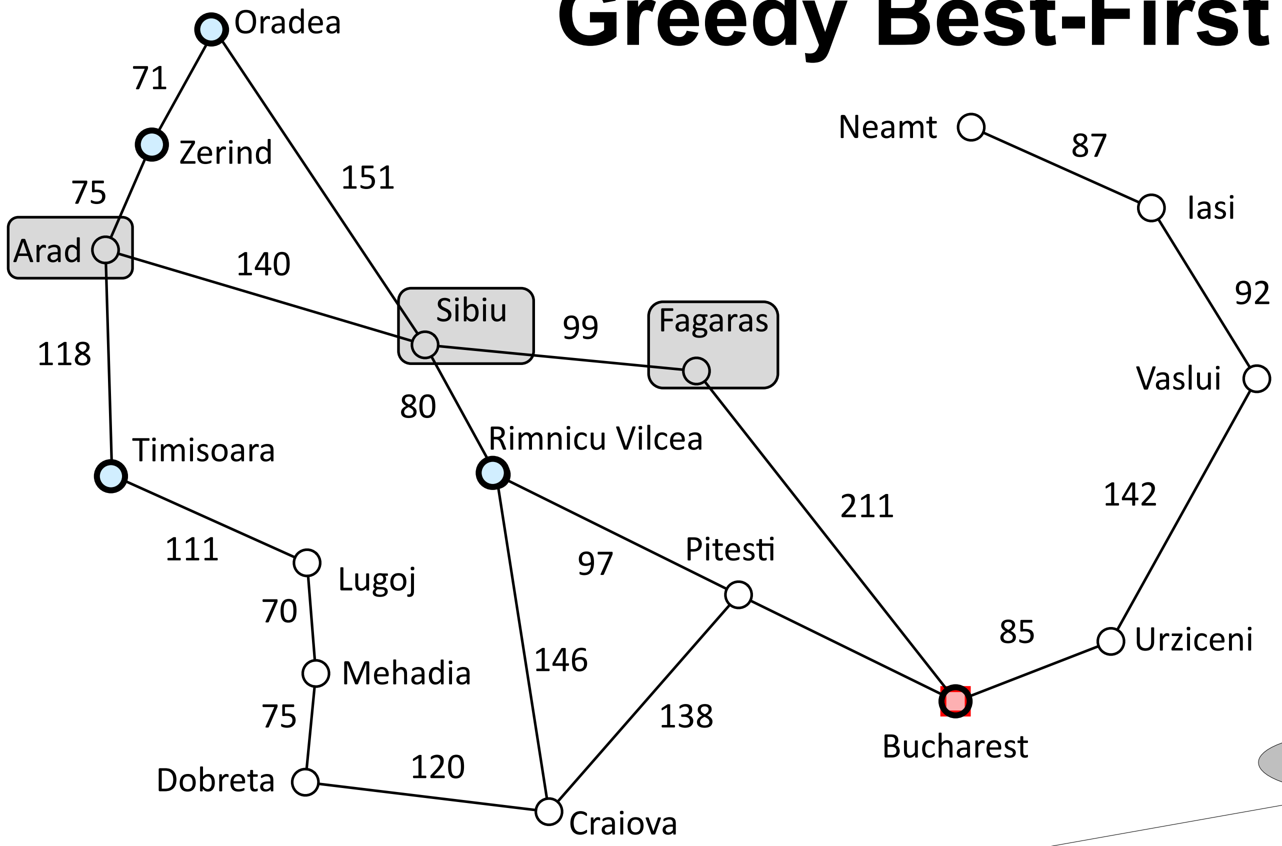
Deterministic-Search(Σ, s_0, g)
  Frontier ← $\{(\langle\rangle, s_0)\}$
  Expanded ← ∅
  while *Frontier* ≠ ∅ do
    select a node $\nu = (\pi, s) \in$ *Frontier* (*i*)
    remove ν from *Frontier*
    add ν to *Expanded*
    if s satisfies g then return π
    Children ←
      $\{(\pi.a, \gamma(s,a))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

- Idea: choose a node that's likely to be close to a goal
- Node selection:
  - Select a node $\nu = (\pi, s) \in$ *Frontier* for which $h(s)$ is smallest
  - Tie-breaking: if more than one such node, choose the oldest
- Pruning: for every node $\nu = (\pi, s)$ in *Children*:
  - If *Children* ∪ *Frontier* ∪ *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

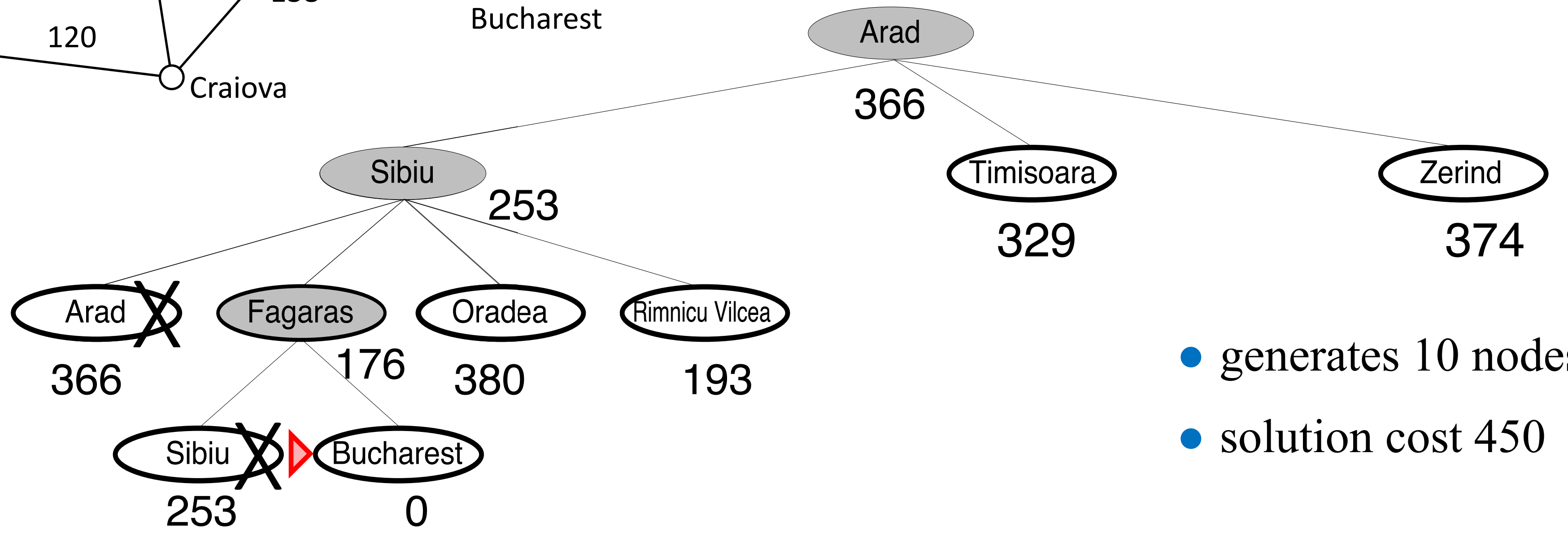
Poll: Have you seen GBFS before?

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

Greedy Best-First Search (GBFS)



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

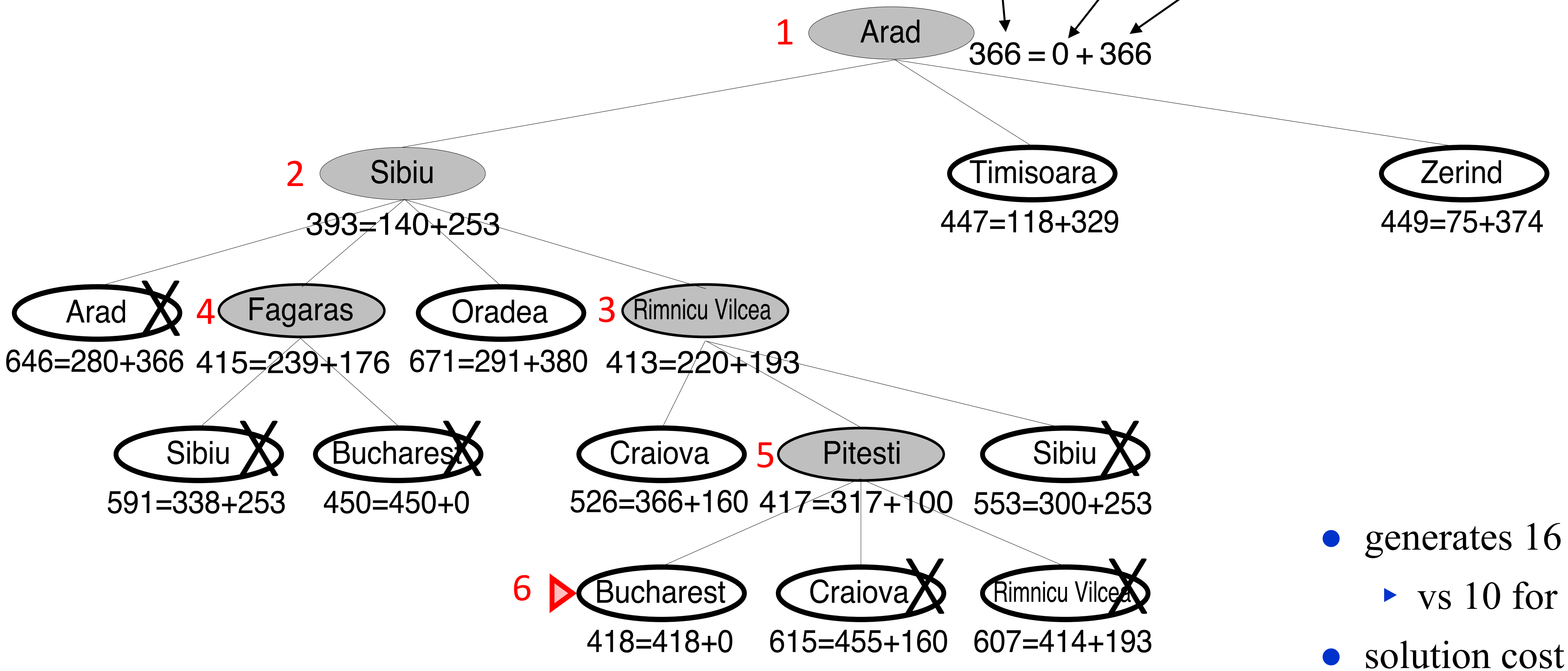
Poll: Have you seen A* before?

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 $f(\nu) = \text{cost}(\pi) + h(s)$
    - Tie-breaking rule: choose oldest
- Pruning: same as in GBFS
  - 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*: under certain conditions (I'll discuss later), can guarantee optimality

$$v = (s,\pi)$$

$$f(v) = cost(\pi) + h(s)$$



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
  - $h^*(s) = \min\{\text{cost}(\pi') \mid \gamma(s,\pi') \text{ satisfies } g\}$
  - $f^*(\nu) = \text{cost}(\pi) + h^*(s)$
  - $f(\nu) = \text{cost}(\pi) + h(s)$

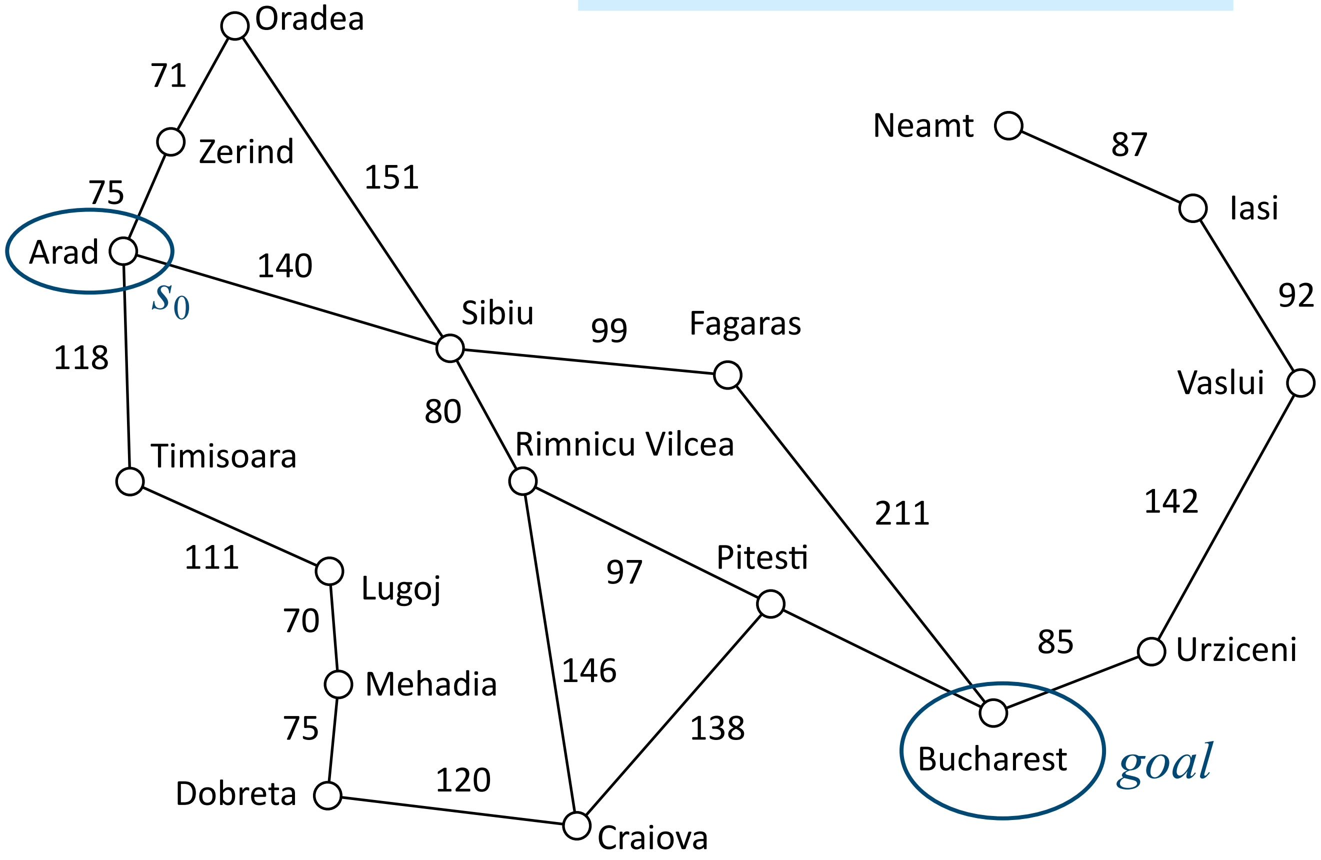
- Definition: h is *admissible* if for every s, $h(s) \leq h^*(s)$

- *Optimality:*
  - in classical planning problems, if h is admissible then any 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:
  - $\nu = (\pi,s)$, where π is the plan for going from s_0 to s
  - $h^*(s) = \min\{\text{cost}(\pi') \mid \gamma(s,\pi') \text{ satisfies } g\}$
  - $f^*(\nu) = \text{cost}(\pi) + h^*(s)$
  - $f(\nu) = \text{cost}(\pi) + h(s)$

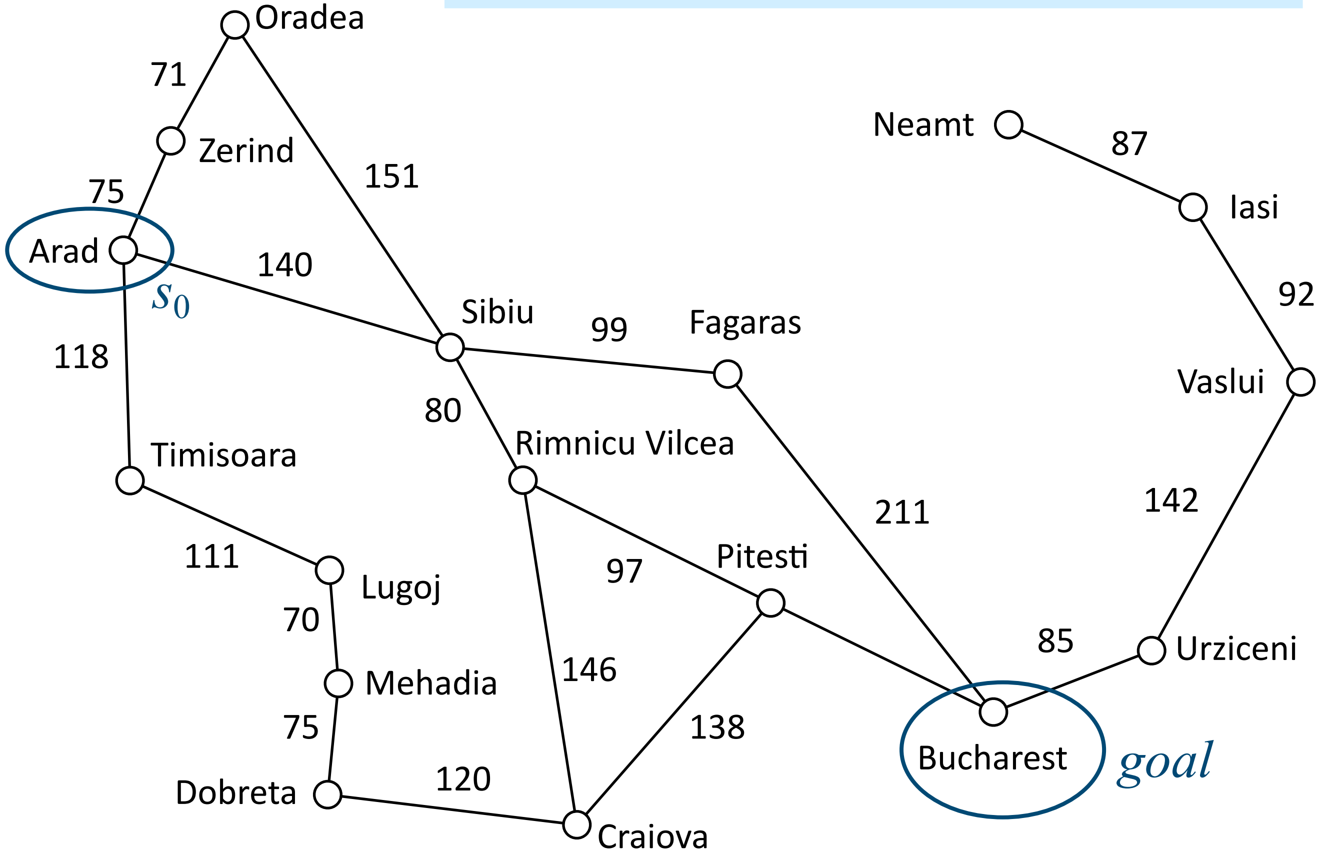
Poll: If h is admissible, does it follow that for every expanded node ν, $f(\nu) \le f^*(\nu)$?

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

A. Yes B. No C. Not sure

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

- *Optimality:*
  - in classical planning problems, if h is admissible then any solution returned by A* will be optimal (least 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

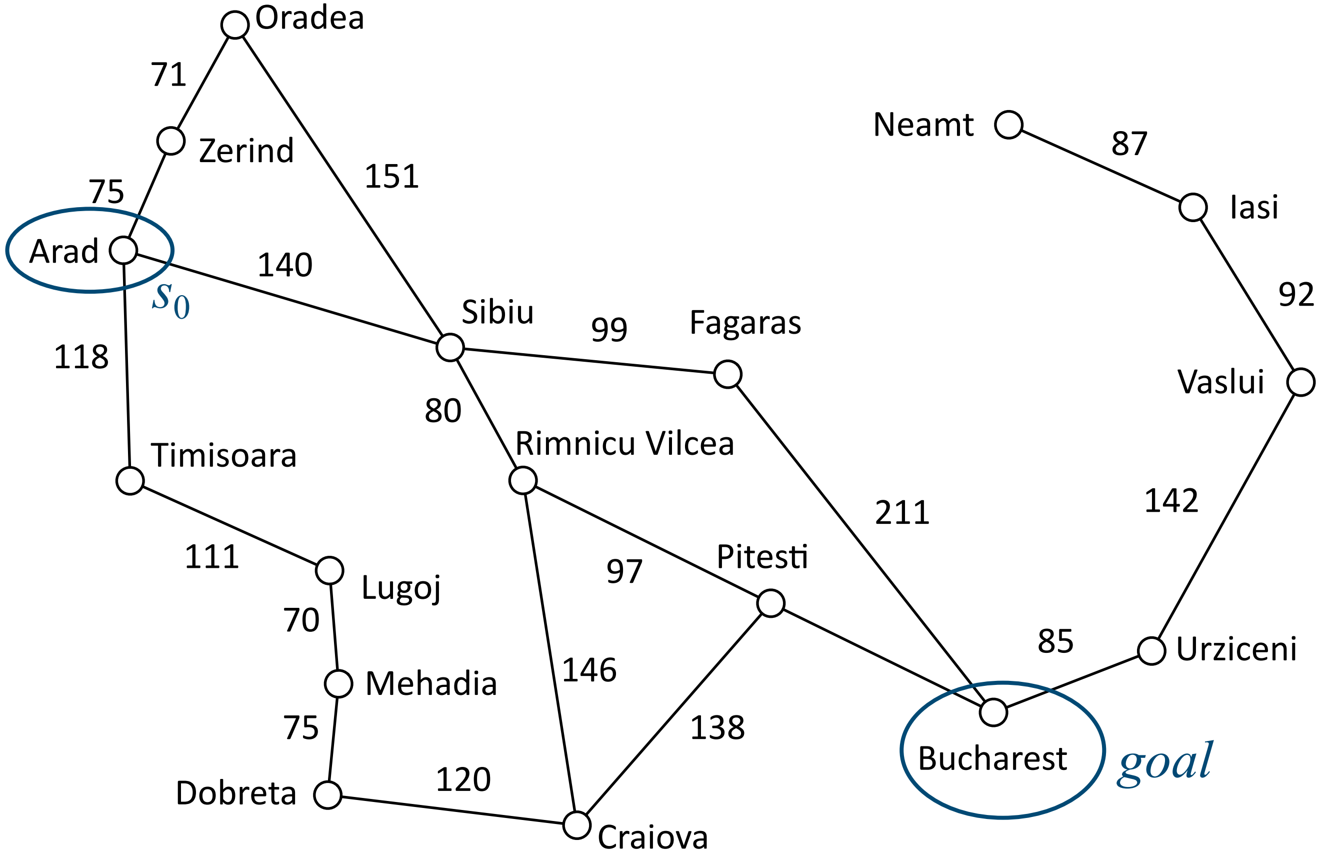
Dominance

- 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

- 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

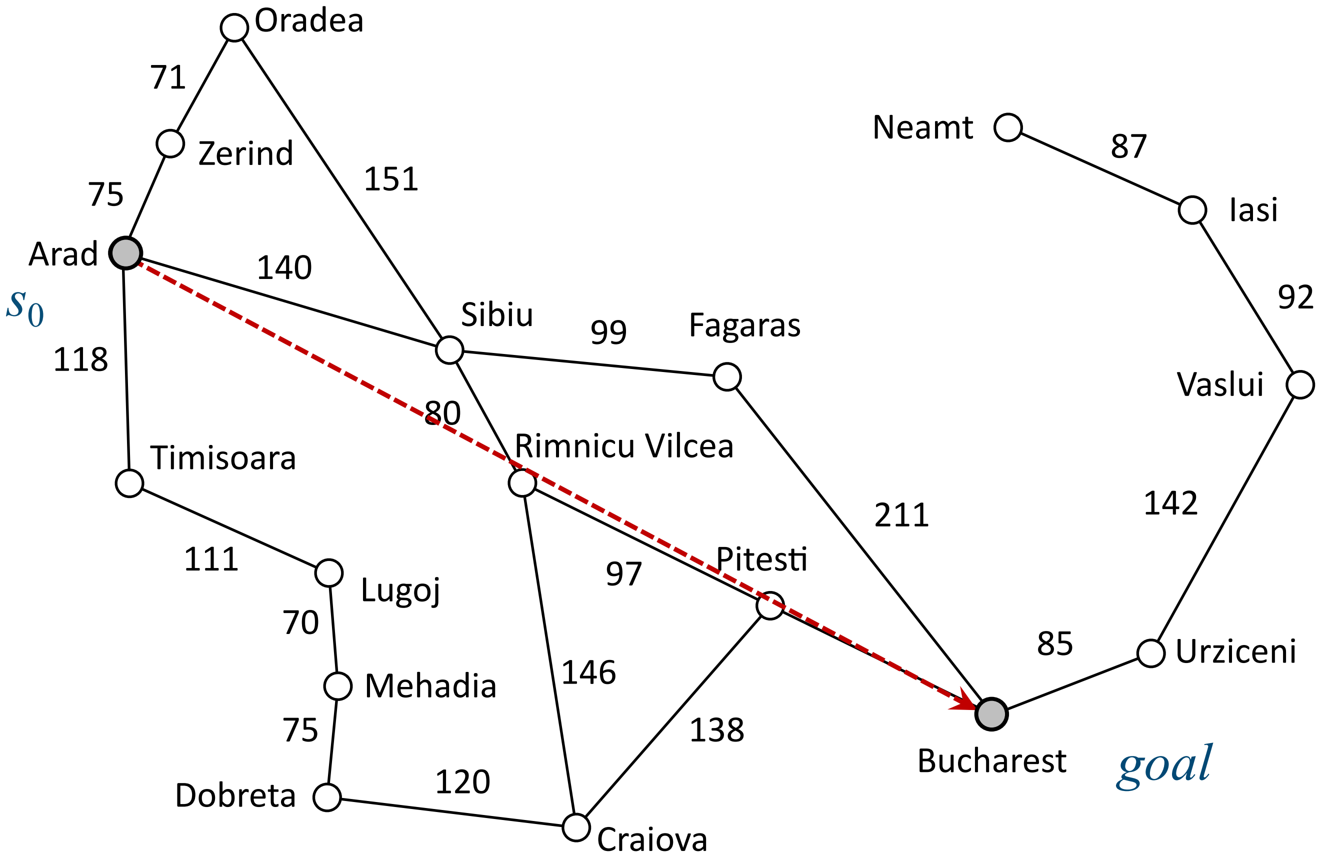


straight-line dist. from s to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Fagaras	176
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

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
  - 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 is like DFS: $O(bl)$ memory, $O(b^l)$ time

Deterministic-Search(Σ, s_0, g)
    Frontier ← $\{(\langle\rangle, s_0)\}$
    Expanded ← ∅
    $c^* \leftarrow \infty$; $\pi^* \leftarrow$ failure
    while *Frontier* ≠ ∅ do
        select a node $\nu = (\pi, s) \in$ *Frontier*   (*i*)
        remove ν from *Frontier* and add it
            to *Expanded*
        ~~if s satisfies g then return π~~
        if s satisfies g and $\text{cost}(\pi) < c^*$ then
            $c^* \leftarrow \text{cost}(\pi)$; $\pi^* \leftarrow \pi$
        else if $f(\nu) < c^*$ then
            Children ←
                $\{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies pre}(a)\}$
            prune 0 or more nodes from
                Children, *Frontier*, *Expanded*   (*ii*)
            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 express as modified version of Deterministic-Search
- Node (step *i*) selection like DFS:
  - Select $\nu = (\pi, s) \in$ *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 $f(\nu) \geq c^*$ then discard ν

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)

IDS(Σ, s_0, g)

for $k = 1$ to ∞ do

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

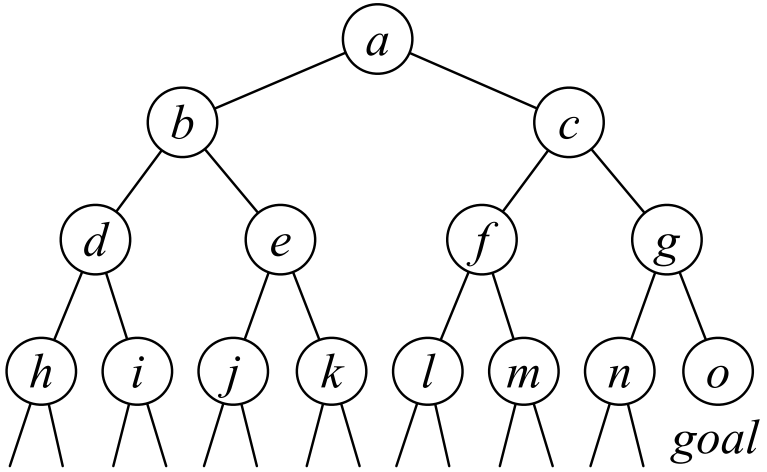
if the search found a solution then return it

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 ⟨*a,c,g,o*⟩
- Total number of nodes generated:
  - 3+7+15 = 25

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

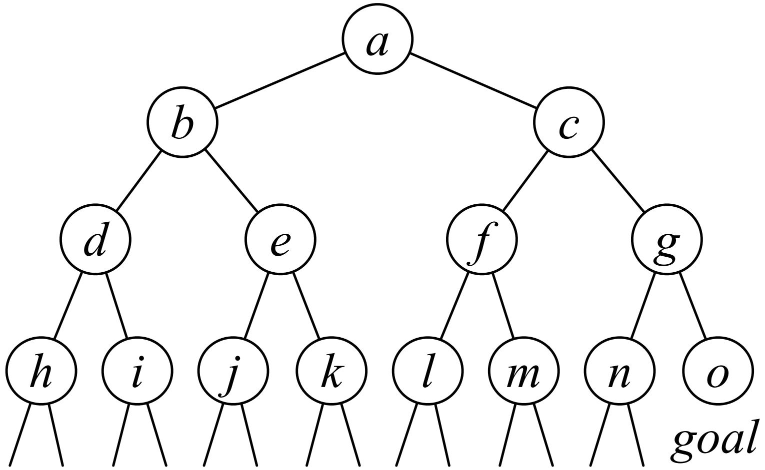
A. $O(b2^d)$
B. $O((b/2)^d)$
C. $O(b^d)$
D. $O(b^{d+1})$
E. something else

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

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

Summary

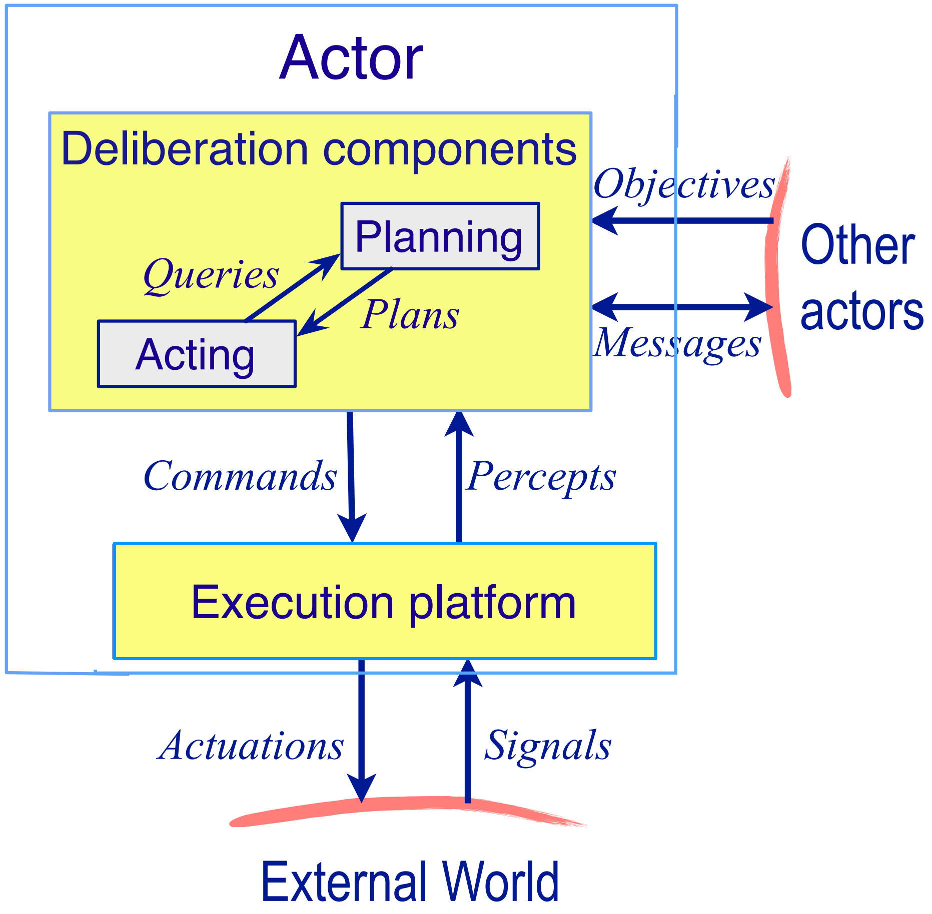
- 2.2 Forward State-Space Search
  - Forward-search, Deterministic-Search
  - cycle-checking
  - Breadth-first, depth-first, uniform-cost search
  - A*, GBFS
  - DFBB, IDS

Outline

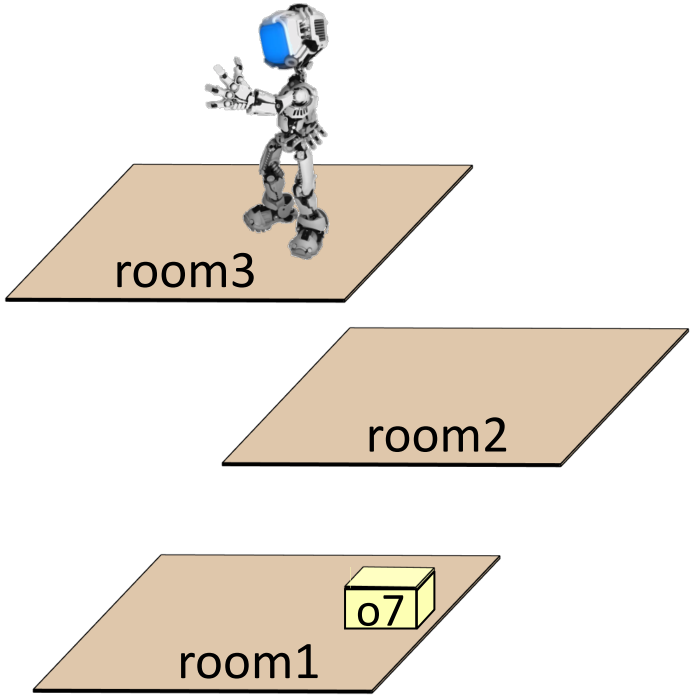
2.6 Incorporating Planning into an Actor

- 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 $\langle s_0, a_1, s_1, a_2, s_2, \ldots\rangle$
- Most real-world environments don't satisfy the assumptions

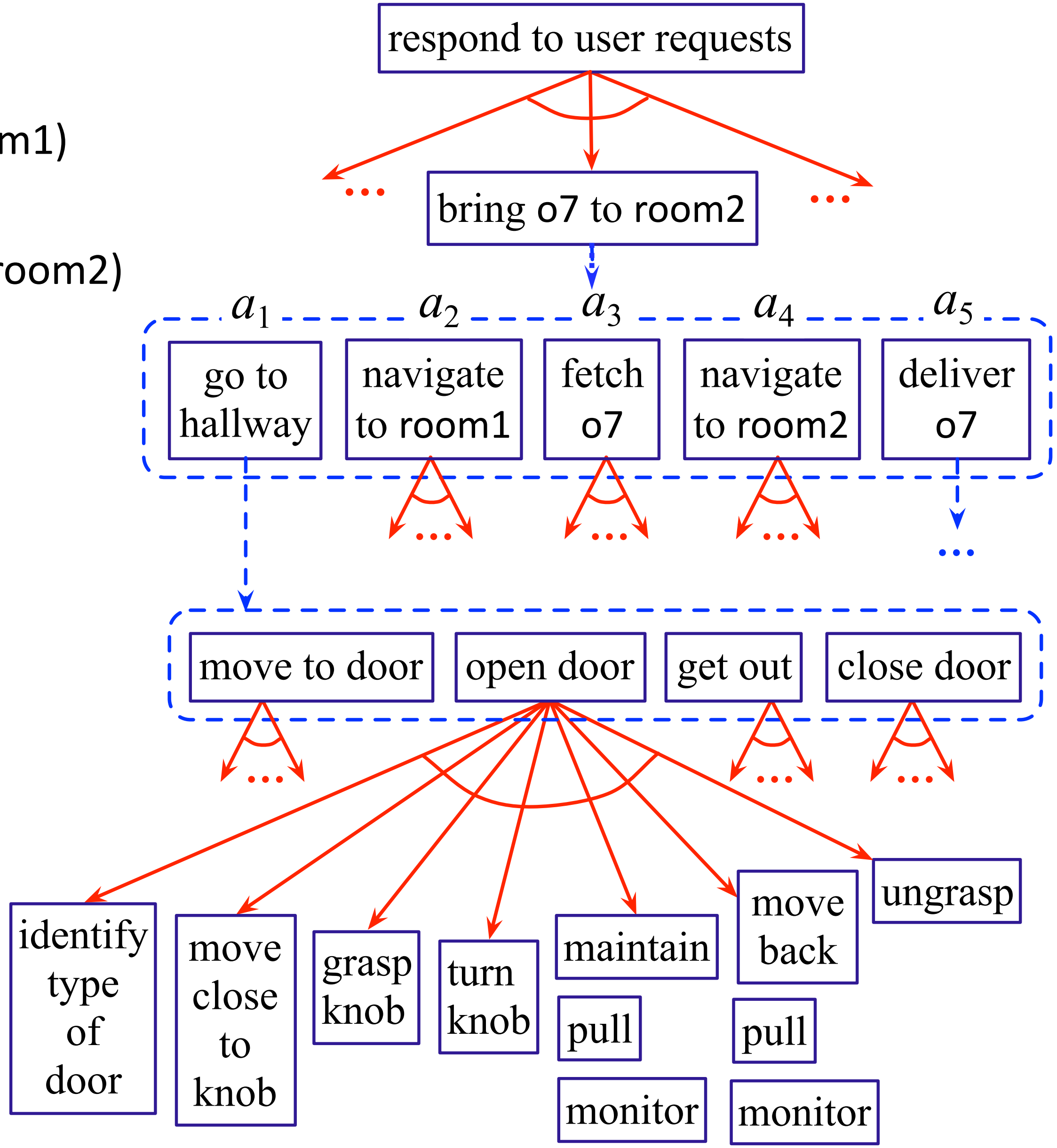
  ⇒ Errors in prediction
- OK if
  - errors occur infrequently, and
  - they don't have severe consequences
- What to do if an error *does* occur?



Service Robot



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)⟩



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

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

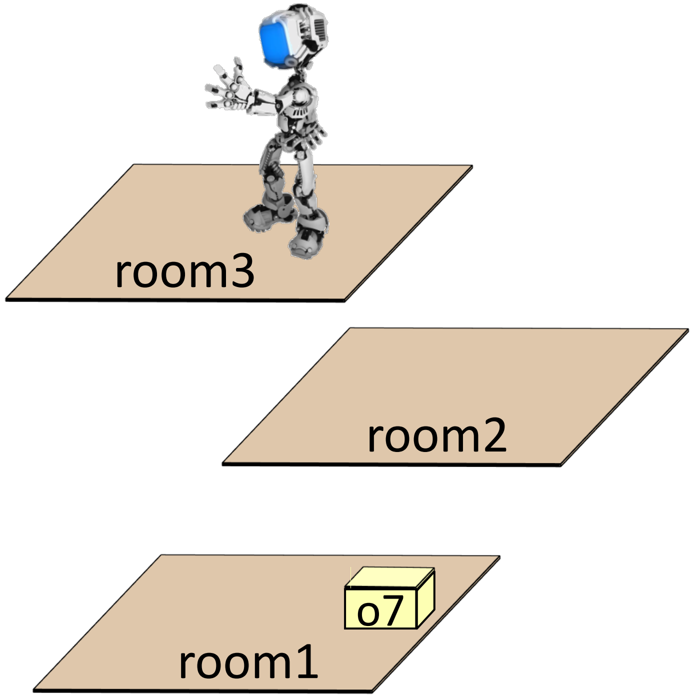
navigate(r,l,m)
- pre: ¬adjacent(l, m), loc(r)=l
- eff: loc(r) ← m

ignores how do navigation, localization

take(r,l,o)
- pre: loc(r)=l, loc(o)=l, cargo(r)=nil
- eff: loc(o) ← r, cargo(r) ← o

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

Service Robot



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)⟩

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

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

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

ignores how do navigation, localization

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 find o, get access to it, grasp it, lift it

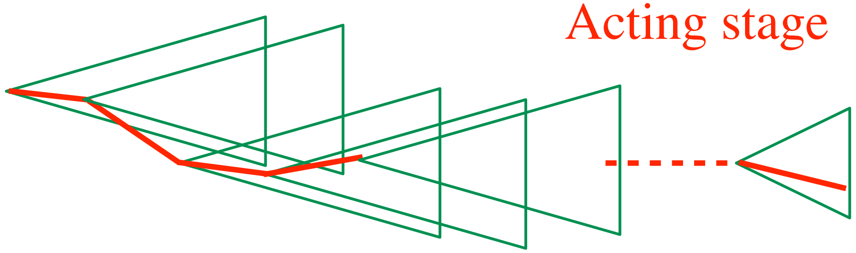
- Some things that can go wrong:
  - *Execution failures*
    - locked door
    - robot battery goes dead
  - *Unexpected events*
    - class ends, hallway gets crowded
    - hallway closed for maintenance
  - *Incorrect information*
    - navigation error, go to wrong place
  - *Missing information*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

Using Planning in Acting

the planner

```
Run-Lookahead(Σ, g)
    s ← abstraction of observed state ξ
    while s ⊭ g do
        π ← Lookahead(Σ, s, g)
        if π = failure then return failure
        a ← pop-first-action(π); perform(a)
        s ← abstraction of observed state ξ
```

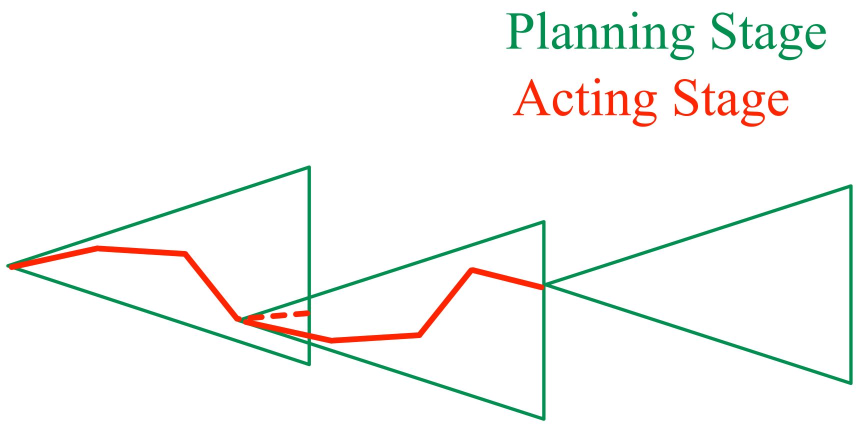
Planning stage
Acting stage



- Call Lookahead, obtain π, perform 1st 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 ξ changes during the wait?

Using Planning in Acting

```
Run-Lazy-Lookahead(Σ, g)
    s ← abstraction of observed state ξ
    until s satisfies g do
        π ← Lookahead(Σ, s, g)
        if π = failure then return failure
        until π = ⟨ ⟩ or s ⊨ g or Simulate(Σ, s, g, π) = failure do
            a ← pop-first-action(π); perform(a)
            s ← abstraction of observed state ξ
```

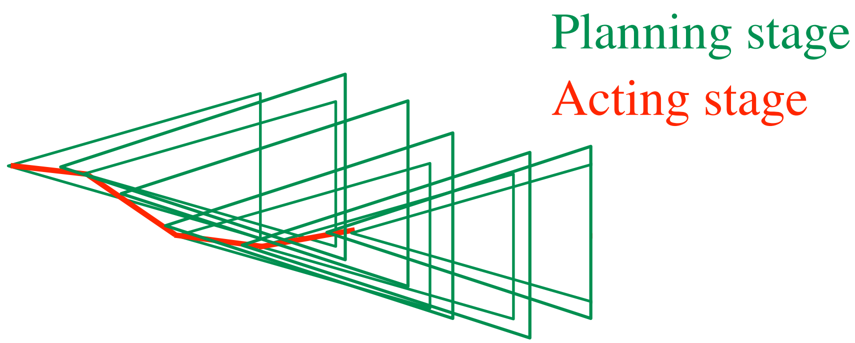


- Call Lookahead, execute the plan as far as possible, don't call Lookahead again unless necessary
- Simulate tests whether the plan will execute correctly
  - Lower-level refinement, physics-based simulation
- What if you don't have a simulation program?
  - Could write Simulate(…) to test whether $\gamma(s,\pi)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

Using Planning in Acting

Run-Concurrent-Lookahead (basic idea)

- global s, π
- thread 1:
  - loop:
    - $s \leftarrow$ observed state
    - $\pi \leftarrow$ Lookahead(Σ,s,g)
- thread 2:
  - loop:
    - $a \leftarrow$ pop-first-element(π)
    - perform a
    - return if observed state $\vDash g$

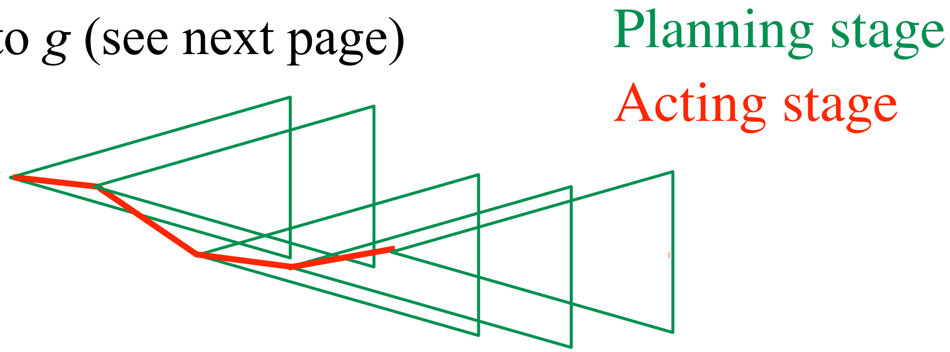


- Motivation: plan and act in a dynamically changing environment
  - Want a recent plan, rather than the old one that Run-Lazy-Lookahead would use
  - Want to get it quickly, rather than waiting like Run-Lookahead
- But there are several problems with the pseudocode
  - It ignores some implementation details
    - how to do locking
    - whether each thread has correct values for π 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 Lookahead(Σ, $\gamma(s,a)$, g)

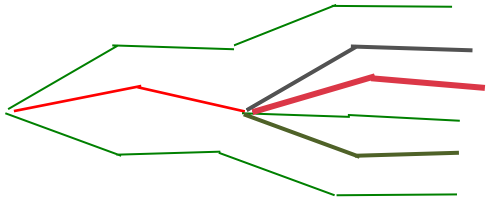
How to do Lookahead

Some possibilities (can also combine these)

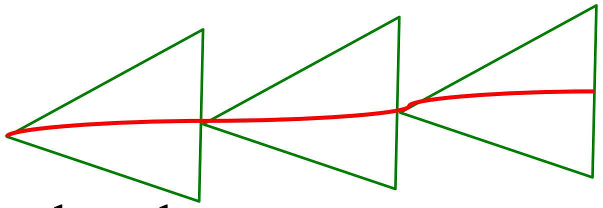
- **Full planning** (if the planner can solve the planning problem quickly enough)
- **Receding horizon**
  - Modify Lookahead to search just part of the way to *g* (see next page)
  - E.g., cut off search when one of the following exceeds a maximum threshold:
    - plan length, plan cost, computation time



- **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)



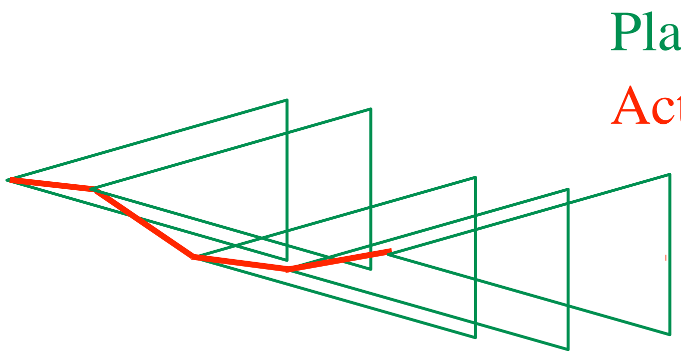
- **Subgoaling**
  - Tell Lookahead to plan for some subgoal g_1, rather than g itself
  - Once the actor has achieved g_1, tell Lookahead to plan for the next subgoal g_2
  - And so forth until the actor reaches g



Receding-Horizon Search

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

- Lookahead = modified version of Deterministic-Search
  - Before line (*i*), put something like one of these:
    - *time-based cutoff*: if time-left() = 0 then return π
    - *length-based cutoff*: if $|\pi| > l_{max}$ then return π
    - *cost-based cutoff*: if $f(\nu) > c_{max}$ then return π
    - *closeness to goal*: if $h(s) \leq \varepsilon$ then return π
  - 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 π^* = least costly partial solution of length $\leq l_{max}$



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