Advanced Topics Data Science and AI

Automated Planning and Acting

Deterministic Models

Tanya Braun

UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR INFORMATIONSSYSTEME

Content

1. Planning and Acting with **Deterministic** Models
    a. State-variable representation
    b. Forward State-Search Space
    c. Heuristic Functions
    d. Backward Search
    e. Plan-Space Search
2. Planning and Acting with **Refinement** Methods
3. Planning and Acting with **Temporal** Models
4. Planning and Acting with **Nondeterministic** Models
5. Making Simple Decisions
6. Making Complex Decisions
7. Planning and Acting with **Probabilistic** Models
8. Provably Beneficial AI

- Other: open world, perceiving, learning
    - If time permits

Outline per the Book

2.1 *State-variable representation*
- State = {values of variables}; action = changes to those values

2.2 *Forward state-space search*
- Start at initial state, look for sequence of actions that achieve goal

2.3 *Heuristic functions*
- How to guide a forward state-space search

2.6 *Incorporating planning into an actor*
- Online lookahead, unexpected events

2.4 *Backward search*
- Start at goal state, go backwards toward initial state

2.5 *Plan-space search*
- Start with incomplete plan for getting from initial state to goal state, make transformations to fix flaws in the plan

Motivation

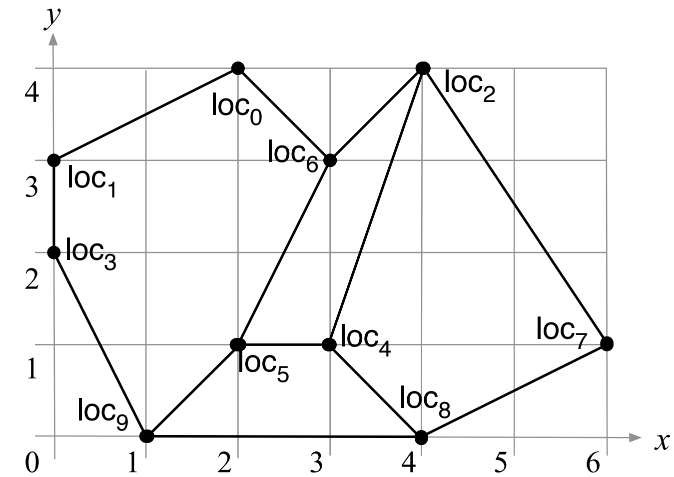
- How to model a complex environment?
  - Generally need simplifying assumptions
- **Classical planning**
  - Finite, static world
    - Change occurs only when the actor causes it
  - No concurrent actions, no explicit time
    - Just a sequence of states and actions $\langle s_0, a_1, s_1, a_2, s_2, \ldots \rangle$
  - Determinism, no uncertainty
    - Can predict exactly what each action will do
    - No accidents, no “chance” outcomes
- Avoids a lot of complications
  - But most real-world environments don’t satisfy the assumptions
- Errors in prediction
  - OK if they’re infrequent and don’t have severe consequences

Domain Model

- State-transition system (or *classical planning domain*)
  - $\Sigma = (S, A, \gamma, cost)$; $cost$ is optional
    - S - finite set of states that the system may be in
    - A - finite set of actions: things the actor can do
    - $\gamma : S \times A \longrightarrow S$ - prediction function (or *state-transition function*)
      - partial function: $\gamma(s, a)$ isn't defined unless a is applicable in s
      - $Dom(a) = \{s \in S \mid \gamma(s, a) \text{ is defined}\} = \{s \in S \mid a \text{ applicable in s}\}$
      - $Range(a) = \{\gamma(s, a) \mid s \in Dom(a)\}$
    - $cost : S \times A \longrightarrow \mathbb{R}^+$ -or- $cost : A \longrightarrow \mathbb{R}^+$
      - Could be monetary cost, time required, something else
      - Often omitted from Σ; default is $cost(a) = 1$
- Classical planning problem: $P = (\Sigma, s_0, S_g)$
  - (planning domain, initial state, set of goal states)
  - $s_0 \in S, S_g \subseteq S$
- Solution for P: a sequence of actions called plan that will produce a state 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



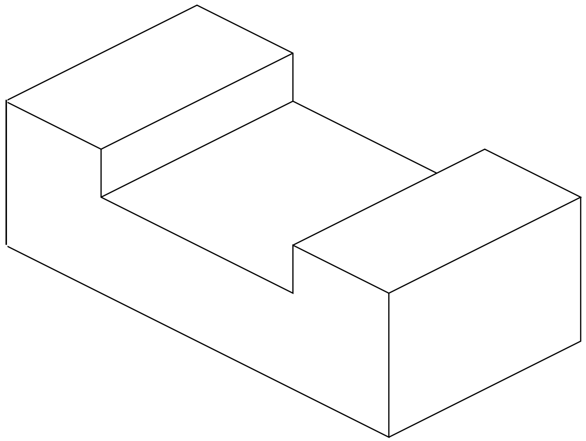
- In larger domains, don't represent all states explicitly
  - Language for describing properties of states
  - Language for describing how each action changes those properties
  - Start with initial state, use actions to produce other states

Domain-specific Representation

- Made to order 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 in a state s
    - 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

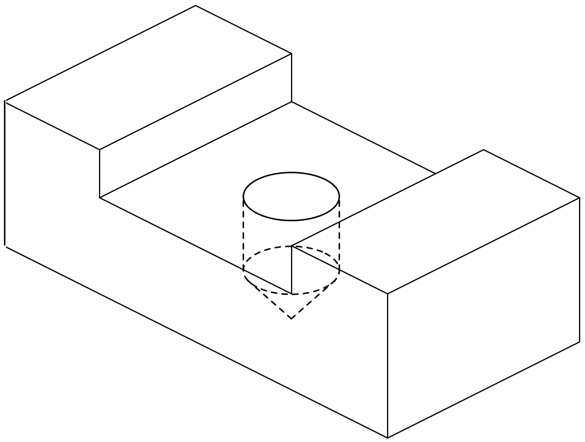
Example

- Drilling holes in a metal workpiece
  - A state
    - Geometric model of the workpiece,
      information about its location and orientation
    - Capabilities and status of drilling machine and drill bit
  - Several actions
    - Putting the workpiece onto the drilling machine
    - Clamping it
    - Loading a drill bit
    - Drilling (next slide)



Drilling

- Name and parameters:
  - drill-hole(*machine, drill-bit, workpiece, geometry, machining-tolerances*)
- Preconditions
  - Can the drilling machine and drill bit produce a hole having the desired geometry and machining tolerances?
  - Is the drill bit installed? Is the workpiece clamped onto the drilling platform? Etc.
- Effects
  - Geometric model of new workpiece geometry, annotated with tolerances
- Cost
  - Estimate of time or monetary cost

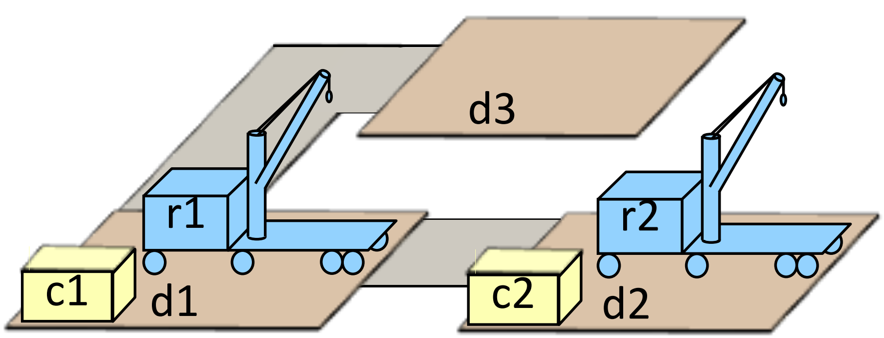


Discussion

- Advantage of domain-specific representation:
  - Can choose 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

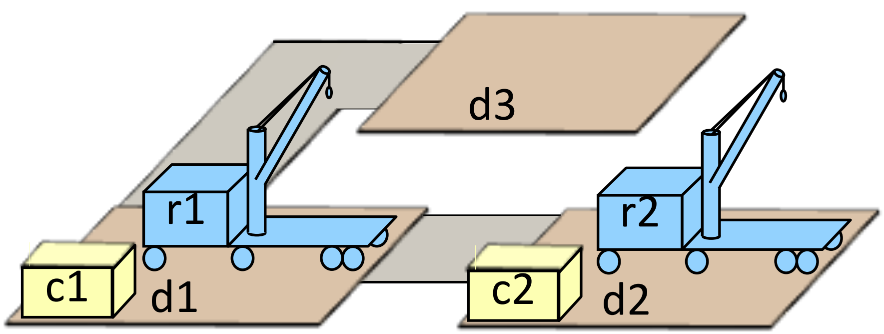
State-Variable Representation

- E : environment that we want to represent
- B : set of objects
  - Names for objects in E, mathematical constants, …
  - Only needs to include objects that matter at current level of abstraction
- Example (slightly different from the book)
  - $B = Robots \cup Containers \cup Locs \cup \{nil\}$
    - $Robots = \{r1, r2\}$
    - $Containers = \{c1, c2\}$
    - $Locs = \{d1, d2, d3\}$
- Can omit lots of details
  - E.g., physical characteristics of robots, containers, loading docks, roads



Properties of Objects

- Define ways to represent properties of objects
  - Two kinds of properties: rigid and varying
  - Sets of rigid properties R and varying properties X
- Rigid property: n-ary relation r over B
  - Stays the same in every state
  - Representation
    - As a mathematical relation
      - $adj \ = \{(d1, d2), (d2, d1), (d1, d3), (d3, d1)\}$
    - As a set of ground atoms
      - $adj(d1, d2), adj(d2, d1), adj\ (d1, d3), adj\ (d3, d1)$

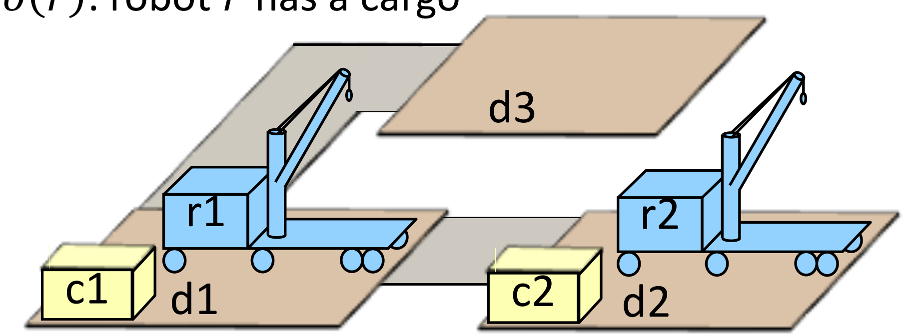


Varying Properties

- Varying property x (or *fluent*)
  - May differ in different states
  - Represent it using a state variable to assign a value to
- Set of state variables $X = \{loc(r1), loc(r2), loc(c1), loc(c2), cargo(r1), cargo(r2)\}$
- Each state variable $x \in X$ has a range

$$\mathcal{R}(x) = \{\text{all values that can be assigned to } x\}$$

  - $\mathcal{R}\big(loc(r1)\big) = \mathcal{R}\big(loc(r2)\big) = Locs$
  - $\mathcal{R}(loc(c1)) = \mathcal{R}(loc(c2)) = Robots \cup Locs$
  - $\mathcal{R}(cargo(r1)) = \mathcal{R}(cargo(r2)) = Containers \cup \{nil\}$
    - $cargo(r)$: robot r has a cargo



States as Functions

- Represent each state as a variable-assignment function
  - Function that maps each $x \in X$ to a value in $\mathcal{R}(x)$

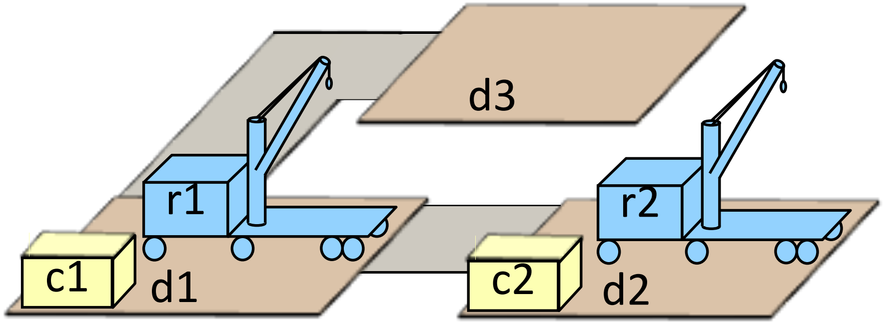
$$s_1(loc(r1)) = d1, \qquad s_1(loc(r2)) = d2,$$
$$s_1(cargo(r1)) = nil, \qquad s_1(cargo(r2)) = nil,$$
$$s_1(loc(c1)) = d1, \qquad s_1(loc(c2)) = d2$$

  - Mathematically, a function is a set of ordered pairs

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

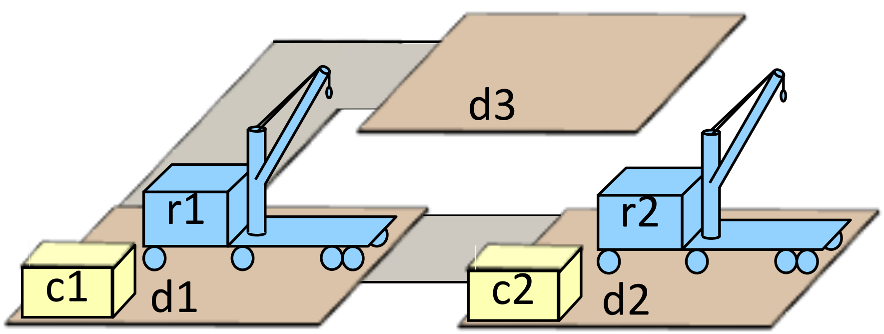
- Write it as a set of ground positive literals (or *ground atoms*):

$$s_1 = \{loc(r1) = d1, \quad cargo(r1) = nil, \quad loc(c1) = d1,$$
$$loc(r2) = d2, \quad cargo(r2) = nil, \quad loc(c2) = d2\}$$



States as Functions

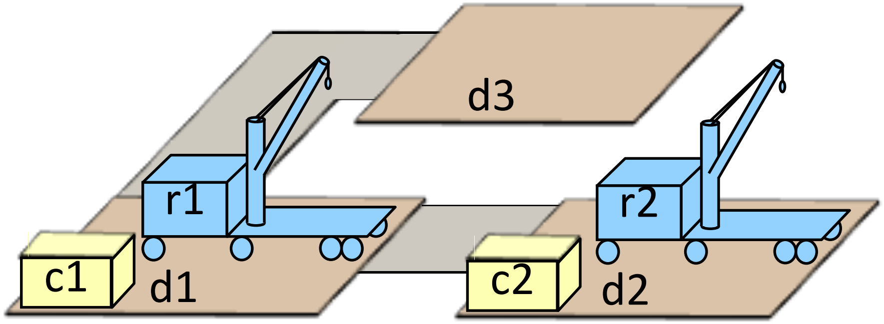
- Let s be a variable-assignment function
  - s is a state only if it has a sensible meaning in our intended environment E
  - Interpretation: a function I
    - Maps each $b \in B$ to an object in E
    - Maps each $r \in R$ to a rigid property in E
    - Maps each $x \in X$ to a varying property in E
- State: a variable-assignment function s such that $I(s)$ can occur in E
  - State space $S = \{\text{all possible states}\}$



Action Templates

- Action template α: a parameterized set of actions
  $\alpha = \big(head(\alpha), pre(\alpha), eff(\alpha), cost(\alpha)\big)$
- $head(\alpha)$: name, parameters
  - Each parameter has a range $\subseteq B$, e.g., $\mathcal{R}(r) = Robots$
- $pre(\alpha)$: precondition literals
  - $rel(t_1, \ldots, t_k)$
  - $var(t_1, \ldots, t_k) = t_0$
  - $\neg\, rel(t_1, \ldots, t_k)$
  - $\neg\, var(t_1, \ldots, t_k) = t_0$
  - Each t_i is a parameter or an element of B
- $eff(\alpha)$: effect literals
  - $var(t_1, \ldots, t_k) \leftarrow t_0$
- $cost(\alpha)$: a number
  - Optional
  - Default = 1

- Example
  - head: $move(r, l, m)$
    - pre: $loc(r) = l, adj(l, m)$
    - eff: $loc(r) \leftarrow m$
  - head: $take(r, l, c)$
    - pre: $cargo(r) = nil, loc(r) = l, loc(c) = l$
    - eff: $cargo(r) \leftarrow c, loc(c) \leftarrow r$
  - head: $put(r, l, c)$
    - pre: $loc(r) = l, loc(c) = r$
    - eff: $cargo(r) \leftarrow nil, loc(c) \leftarrow l$
  - Ranges
    - $\mathcal{R}(r) = Robots = \{r1, r2\}$
    - $\mathcal{R}(l) = \mathcal{R}(m) = Locs = \{d1, d2, d3\}$
    - $\mathcal{R}(c) = Containers = \{c1, c2\}$



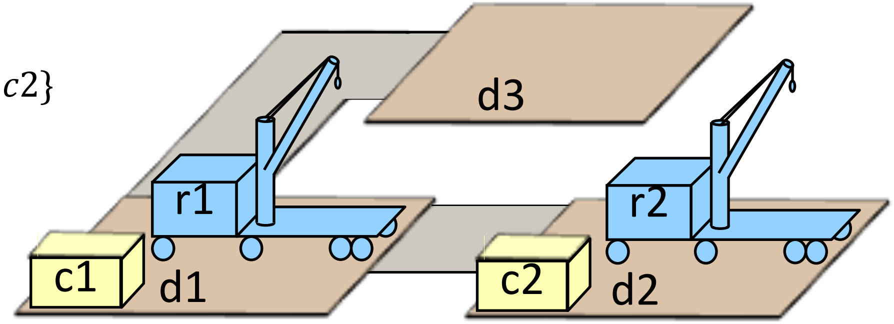
Actions

- $\mathcal{A}$ = set of action templates
- Example $\mathcal{A}$
  - Contains three action templates
  - head: $move(r, l, m)$
    - pre: $loc(r) = l, adj(l, m)$
    - eff: $loc(r) \leftarrow m$
  - head: $take(r, l, c)$
    - pre: $cargo(r) = nil,$ $loc(r) = l, loc(c) = l$
    - eff: $cargo(r) \leftarrow c, loc(c) \leftarrow r$
  - head: $put(r, l, c)$
    - pre: $loc(r) = l, loc(c) = r$
    - eff: $cargo(r) \leftarrow nil, loc(c) \leftarrow l$
  - Ranges
    - $\mathcal{R}(r) = Robots = \{r1, r2\}$
    - $\mathcal{R}(l) = \mathcal{R}(m) = Locs = \{d1, d2, d3\}$
    - $\mathcal{R}(c) = Containers = \{c1, c2\}$

- Action a: *ground instance* of an action template α ∈ $\mathcal{A}$
  - Replace each parameter t occurring in α with a value from $\mathcal{R}(t)$
- Example action:
  - $move(r1, d1, d2)$
    - pre: $loc(r1) = d1,\ adj(d1, d2)$
    - eff: $loc(r1) \leftarrow d2$

How many move actions exist?

- Action space A $= \{\text{all actions we can get from } \mathcal{A}\}$ $= \{\text{all ground instances of elements of } \mathcal{A}\}$

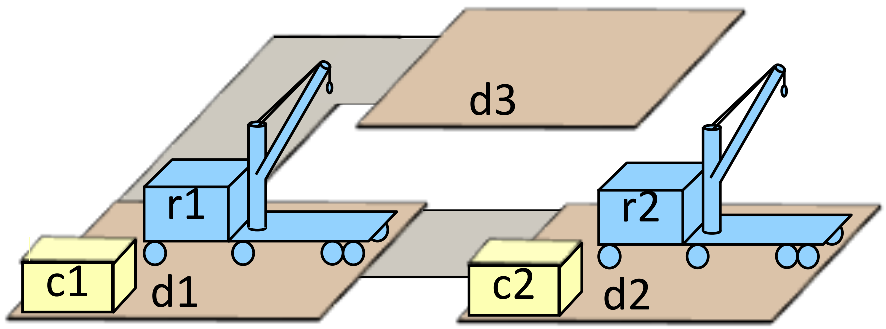


Applicability

- Action a is applicable in state s if
  - For every positive literal $l \in pre(\alpha)$, l is in s or in one of the rigid relations
  - For every negative literal $\neg l \in pre(\alpha)$, l is not in s nor in any rigid relations
- Example
  - Rigid relation: $adj = \{(d1, d2), (d2, d1), (d1, d3), (d3, d1)\}$
  - State $s_1 =$
  $\{cargo(r1) = nil, cargo(r2) = nil,$
  $loc(r1) = d1, loc(r2) = d2,$
  $loc(c1) = d1, loc(c2) = d2\}$
  - Action template $move(r, l, m)$
    - pre: $loc(r) = l, adj(l, m)$
    - eff: $loc(r) \leftarrow m$
  - Ranges
    - $\mathcal{R}(r) = Robots$
    - $\mathcal{R}(l) = \mathcal{R}(m) = Locs$

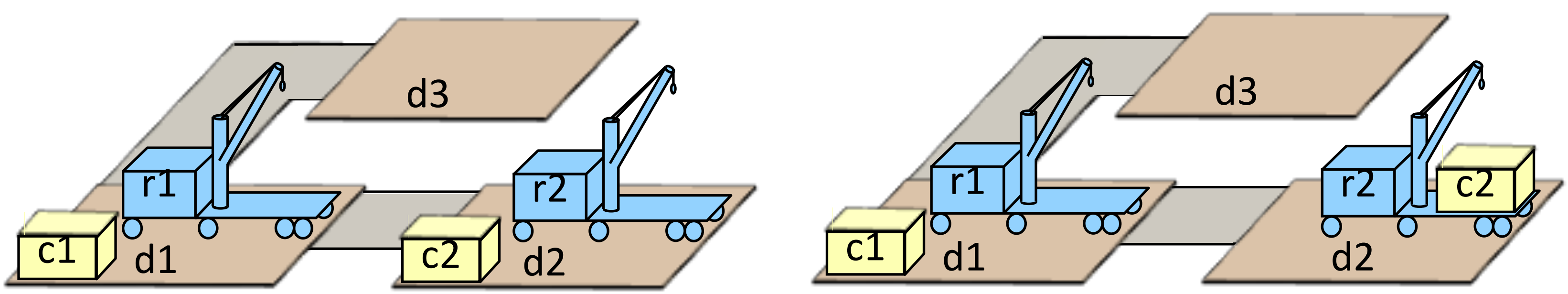
- Applicable action in s_1
  - $move(r1, d1, d2)$
    - pre: $loc(r1) = d1,\ adj(d1, d2)$
    - eff: $loc(r1) \leftarrow d2$

- Not applicable action in s_1
  - $move(r1, d2, d1)$
    - pre: $loc(r1) = d2,\ adj(d2, d1)$
    - eff: $loc(r1) \leftarrow d1$



Computing Prediction Function γ

- If action a is applicable in state s
  - $\gamma(s, a) = \{(x, w) |\ eff(a)$ contains the effect $x \leftarrow w\}$
    $\cup \{(x, w) \in s |\ x$ isn't the target of any effect in $eff(a)\}$
- Example
  - State $s_1 =$
    $\{cargo(r1) = nil, cargo(r2) = nil,$
    $loc(r1) = d1, loc(r2) = d2, \quad loc(c1) = d1, loc(c2) = d2\}$
  - Action $take(r2, d2, c2)$
    - pre: $cargo(r2) = nil, loc(r2) = d2, loc(c2) = d2$
    - eff: $cargo(r2) \leftarrow c2, loc(c2) \leftarrow r2$
  - $s_2 = \gamma\big(s_1, take(r2, d2, c2)\big) =$
    $\{cargo(r1) = nil, cargo(r2) = c2,$
    $loc(r1) = d1, loc(r2) = d2, \quad loc(c1) = d1, loc(c2) = r2\}$



s_1 → s_2

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, $\mathcal{R}(x) \subseteq B$

  S = state space over X
  = {all variable-assignment functions that have sensible interpretations}

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

  - for every parameter t, $\mathcal{R}(t) \subseteq B$

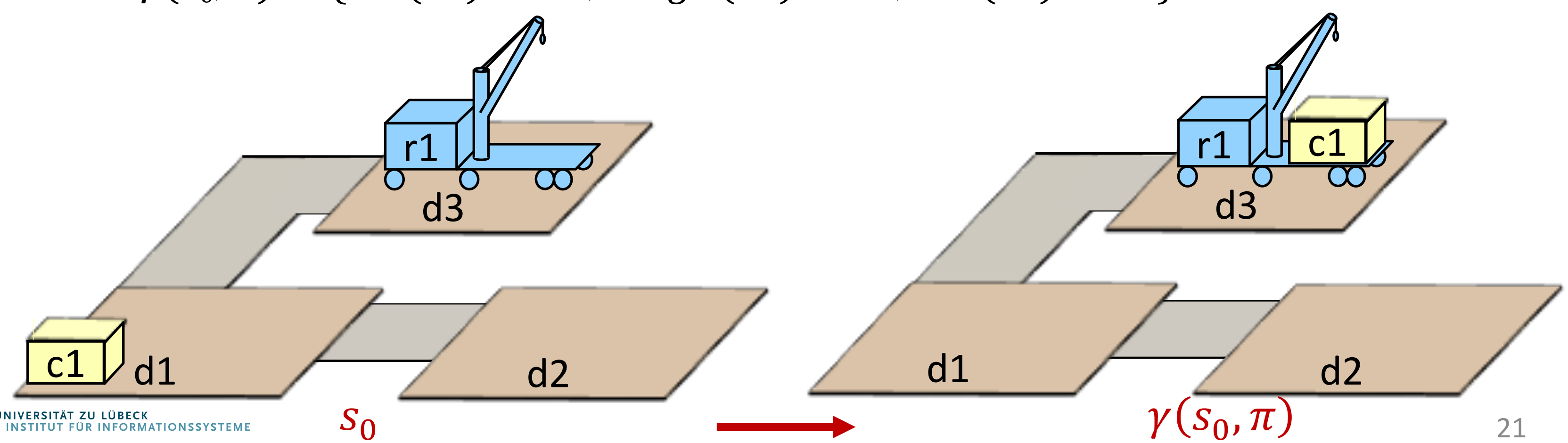
  A = {all ground instances of action templates in $\mathcal{A}$}

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

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

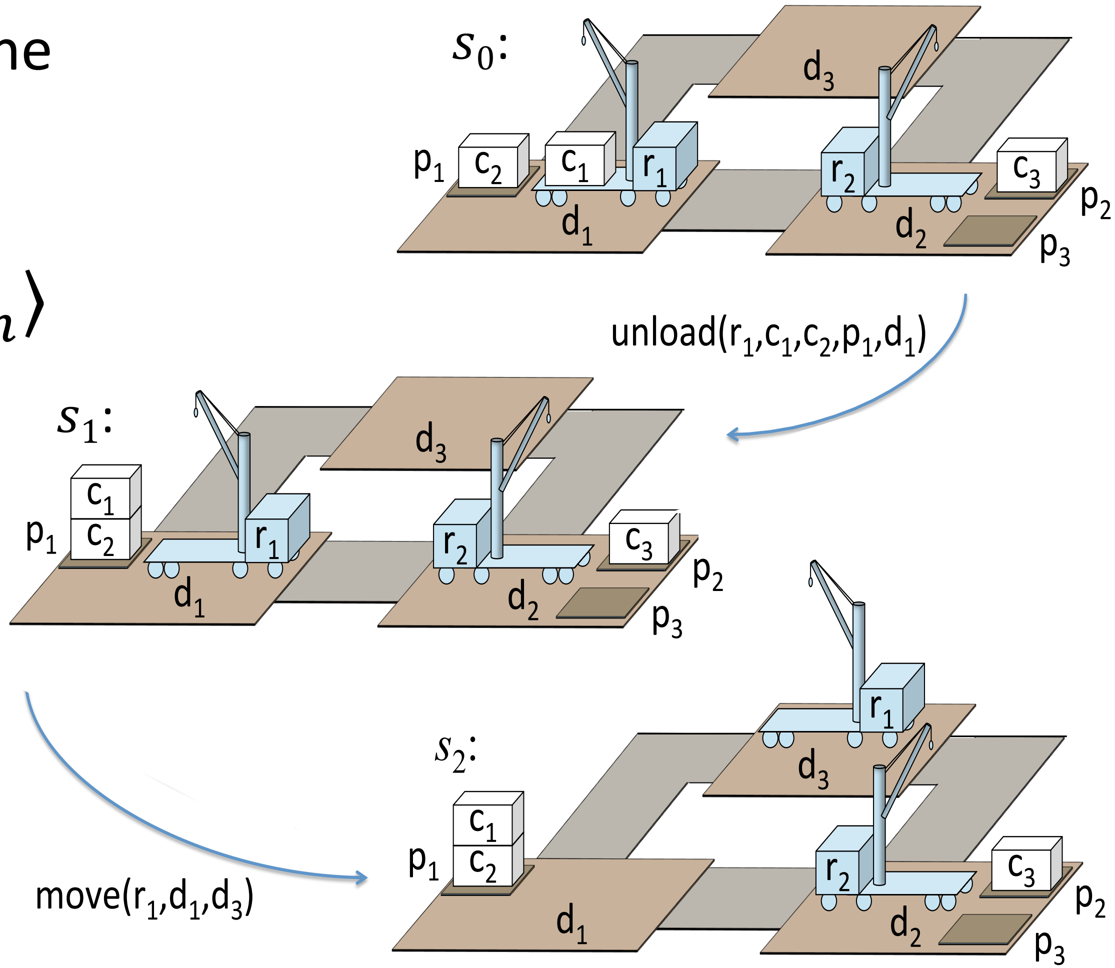
Plans

- Plan: sequence of actions $\pi = \langle a_1, a_2, \ldots, a_n \rangle$
  - $cost(\pi) = \sum_i cost(a_i)$
  - Length of π = n
- π is applicable in s_0 if the actions in π can be applied in the order given,
  - i.e., there are states $s_1, s_2, \ldots, s_n$ such that
    $$\gamma(s_0, a_1) = s_1, \gamma(s_1, a_2) = s_2, \ldots, \gamma(s_{n-1}, a_n) = s_n$$
  - If so, then define $\gamma(s_0, \pi) = s_n$
- Example
  - $s_0 = \{loc(r1) = d3, cargo(r1) = nil, loc(c1) = d1\}$
  - $\pi = \langle move(r1, d3, d1), take(r1, d1, c1), move(r1, d1, d3) \rangle$
    - $cost(\pi) = 3$ (default)
  - $\gamma(s_0, \pi) = \{loc(r1) = d3, cargo(r1) = c1,\ loc(c1) = r1\}$



State Space

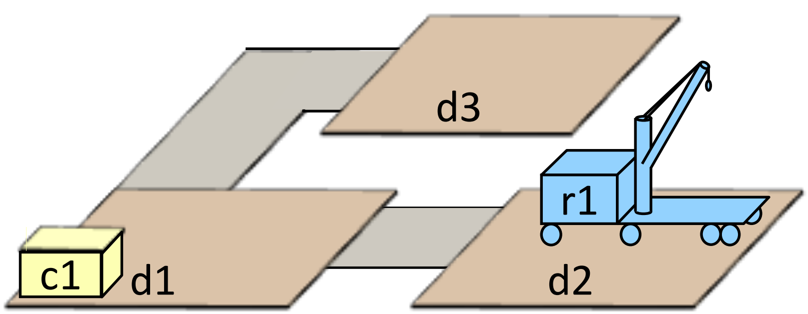
- Directed graph
  - Nodes = states of the world
  - Edges: γ
- If $\pi = \langle a_1, a_2, \dots, a_n \rangle$ is applicable in s_0, it produces a path $\langle s_1, s_2, \dots, s_n \rangle$
  - $\gamma(s_0, a_1) = s_1$, $\gamma(s_1, a_2) = s_2$, … , $\gamma(s_{n-1}, a_n) = s_n$



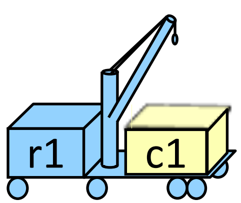
Planning Problems

- State-variable planning problem $P = (\Sigma, s_0, g)$
  - State-variable representation of a classical planning problem
  - $\Sigma = (S, A, \gamma)$ 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 $\gamma(s_0, \pi) \in S_g$, then π is a solution for P
- Example
  - $s_0 = \{loc(r1) = d2, cargo(r1) = nil, loc(c1) = d1\}$
  - $adj = \{(d1, d2), (d2, d1), (d1, d3), (d3, d1)\}$
  - $g = \{cargo(r1) = c1\}$
  - $\pi = \langle move(r1, d2, d1), take(r1, d1, c1) \rangle$

How many solutions of length 3 exist?



s_0



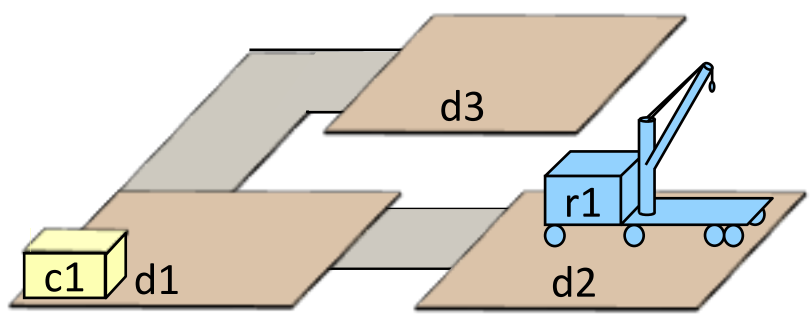
g

Classical Representation

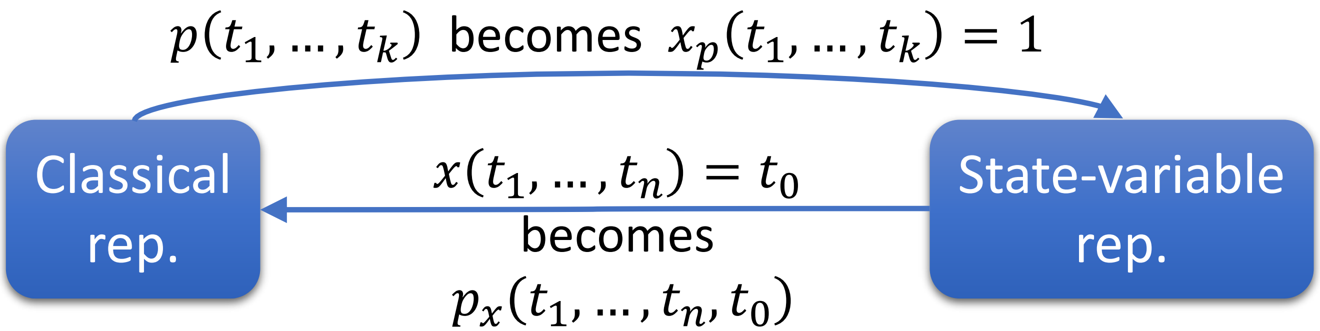
- Motivation
  - The field of AI planning started out as automated theorem proving
  - It still uses a lot of that notation
- Classical representation is equivalent to state-variable representation
  - Represents both rigid and varying properties using logical predicates
    - $adj(l, m)$ - location l is adjacent to location 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
    - $cargo(r) = c \longrightarrow loaded(r)$ - robot r is loaded with a container

Why?

- State s = a set of ground atoms
  - $s_0 = \{adj(d1, d2), adj(d2, d1), adj(d1, d3), adj(d3, d1), loc(c1, d1), loc(r1, d2)\}$



Classical Representation



- Equivalent expressive power
  - Each can be converted to the other in linear time and space

    $\longrightarrow$ Each logical atom p is translated into a Boolean state variable x_p with parameter list $(t_1, \dots, t_k)$
      - Positive literals $p(t_1, \dots, t_k)$ become $x_p(t_1, \dots, t_k) = 1$
      - Negative literals $\neg p(t_1, \dots, t_k)$ become $x_p(t_1, \dots, t_k) = 0$

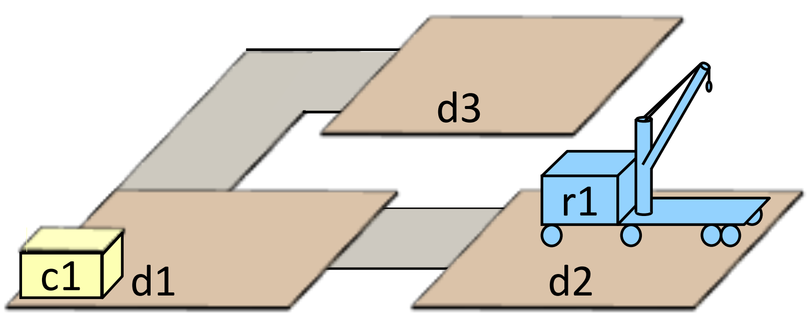
    $\longleftarrow$ Each state variable $x(t_1, \dots, t_k)$ is translated into a set of logical atoms
    $$\{p_x(t_1, \dots, t_k, v) \mid v \in \mathcal{R}(x)\}$$
    - Planning operator $\triangleq$ action template (next slide)
- Worst case complexity: EXPSPACE
  - Time needed to solve a classical planning problem may be exponential in the size of the problem description

Classical Planning Operators

- Operator $o = (head(o), pre(o), eff(o))$
  - $pre(o), eff(o)$ are sets of literals
- Action: a ground instance
- Translation from α to o
  - Precondition $x(t_1, \dots, t_k) = v$
    - $p_x(t_1, \dots, t_k, v)$
  - Precondition $x(t_1, \dots, t_k) \neq v$
    - $\neg p_x(t_1, \dots, t_k, v)$
  - Effect $x(t_1, \dots, t_k) \leftarrow v'$
    - $p_x(t_1, \dots, t_k, v')$
    - If $p_x(t_1, \dots, t_k, v) \in pre(o)$ for some v:
      - Add new effect $\neg p_x(t_1, \dots, t_k, v)$
    - Otherwise
      - Add new parameter u to $head(o)$
      - Add new precondition $p_x(t_1, \dots, t_k, u)$
      - Add new effect $\neg p_x(t_1, \dots, t_k, u)$
- May have twice as many effects and parameters as action template
  - From operator to template: same number

Why?

- Action templates
  - head: $move(r, l, m)$
    - pre: $loc(r) = l, adj(l, m)$
    - eff: $loc(r) \leftarrow m$
  - head: $take(r, l, c)$
    - pre: $cargo(r) = nil, loc(r) = l, loc(c) = l$
    - eff: $cargo(r) \leftarrow c, loc(c) \leftarrow r$
  - head: $put(r, l, c)$
    - pre: $loc(r) = l, loc(c) = r$
    - eff: $cargo(r) \leftarrow nil, loc(c) \leftarrow l$
- Classical planning operators
  - head: $move(r, l, m)$
    - pre: $loc(r, l), adj(l, m)$
    - eff: $\neg loc(r, l), loc(r, m)$
  - head: $take(r, l, c)$
    - pre:$\neg loaded(r), loc(r, l), loc(c, l)$
    - eff: $loaded(r), loc(c, r), \neg loc(c, l)$
  - head: $put(r, l, c)$
    - pre: $loc(r, l), loc(c, r)$
    - eff:$\neg loaded(r), loc(c, l), \neg loc(c, r)$

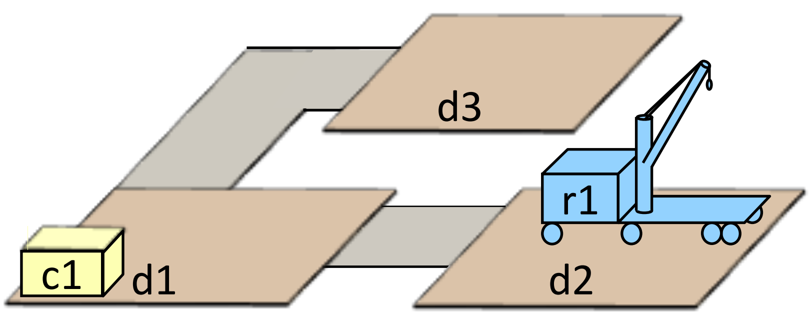


PDDL

- Language for defining planning domains and problems
  - LISP-like syntax
- Original version ≈ 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 planning operators
  - $move(r, l, m)$
    - pre: $loc(r, l), adj(l, m)$
    - eff: $\neg loc(r, l), loc(r, m)$
  - $take(r, l, c)$
    - pre: $\neg loaded(r), loc(r, l), loc(c, l)$
    - eff: $loaded(r), loc(c, r), \neg loc(c, l)$
  - $put(r, l, c)$
    - pre: $loc(r, l), loc(c, r)$
    - eff: $\neg loaded(r), loc(c, l), \neg loc(c, r)$

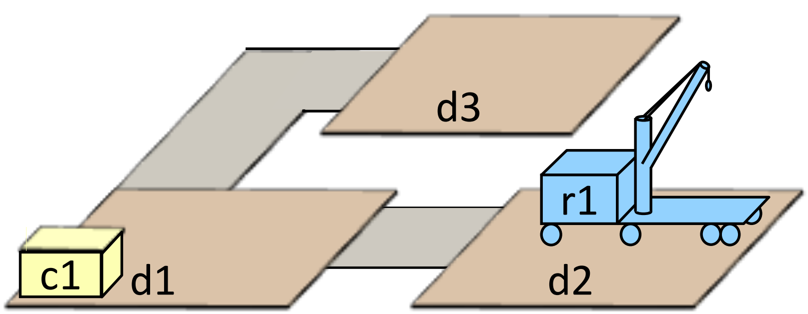



```
(define (domain example-domain-1)
    (requirements :negative-preconditions)

    (:action move
     :parameters (?r ?l ?m)
     :precondition (and (loc ?r ?l)
                        (adj ?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 ?r ?l))
                  (loc ?r ?m)))

    (: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)))))
```

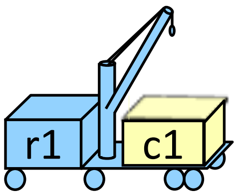
Example problem

- Initial state $s_0 =$
  $\{adj(d1, d2), adj(d2, d1),$
  $adj(d1, d3), adj(d3, d1),$
  $loc(c1, d1), loc(r1, d2)\}$
- Goal state $g = \{loc(c1, r1)\}$

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

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

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



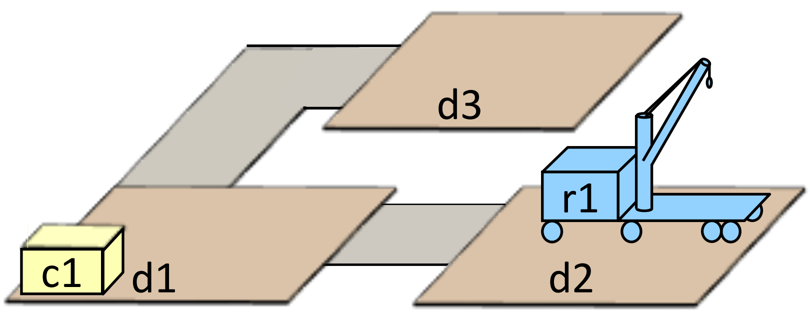
s_0



g

Example typed domain

```
(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))
```

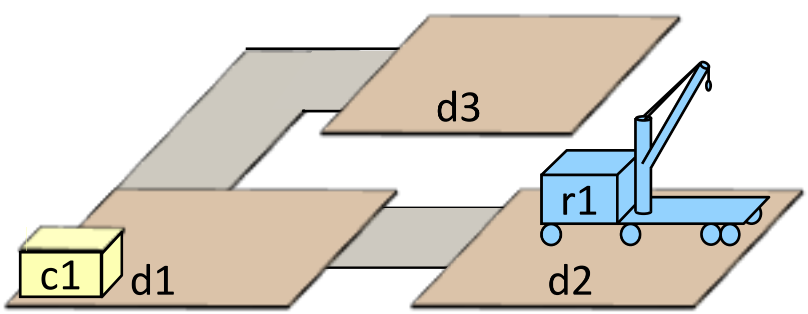



```
(:action move
 :parameters (?r - robot
              ?l ?m - location)
 <<as before>>

(:action take
 :parameters (?r - robot
              ?l - location
              ?c - container)
 <<as before>>

(:action put
 :parameters (?r - robot
              ?l - location
              ?c - container)
 <<as before>>
```

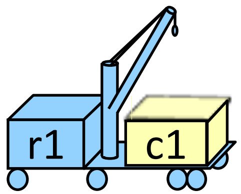
Example typed problem

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

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

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

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

    (:objects
        r1 - robot
        c1 - container
        loc1 loc2 loc3 - location)

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

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



s_0



g

Intermediate Summary

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

Outline per the Book

2.1 *State-variable representation*

- State = {values of variables}; action = changes to those values

2.2 *Forward state-space search*

- Start at initial state, look for sequence of actions that achieve goal

2.3 *Heuristic functions*

- How to guide a forward state-space search

2.6 *Incorporating planning into an actor*

- Online lookahead, unexpected events

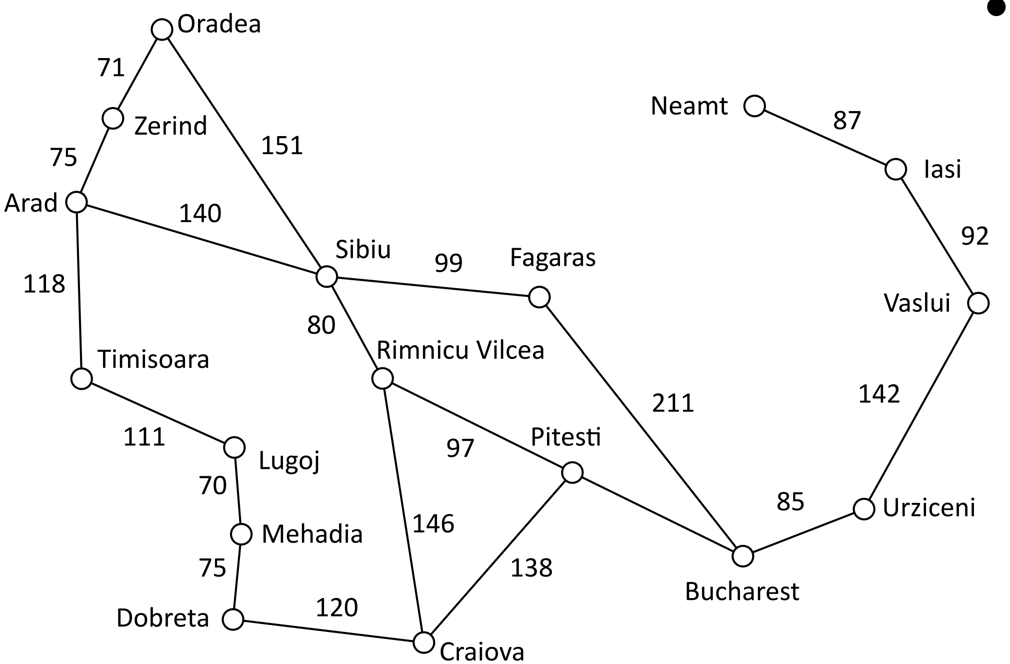
2.4 *Backward search*

- Start at goal state, go backwards toward initial state

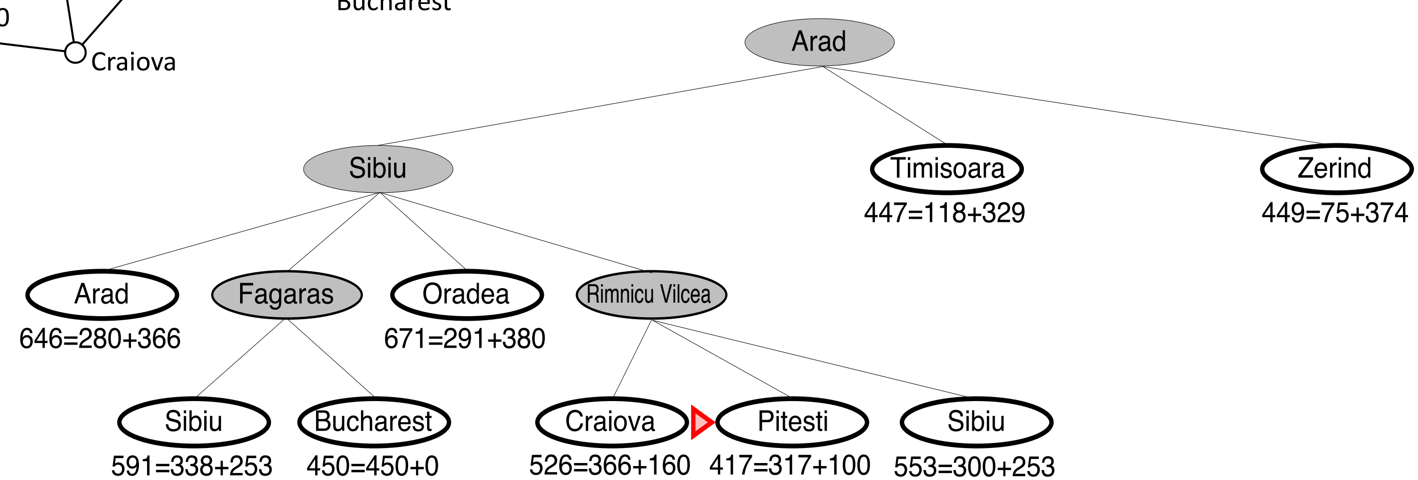
2.5 *Plan-space search*

- Start with incomplete plan for getting from initial state to goal state, make transformations to fix flaws in the plan

Planning as Search



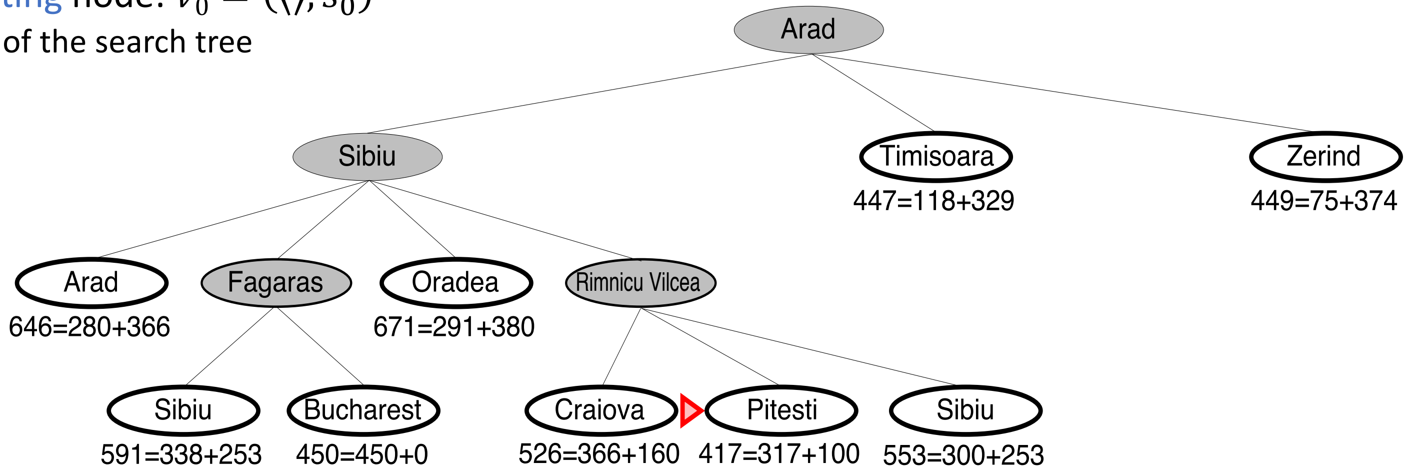
- Nearly all planning procedures are search procedures
  - Search tree: the data structure the procedure uses to keep track of which paths it has explored



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

Search-Tree Terminology

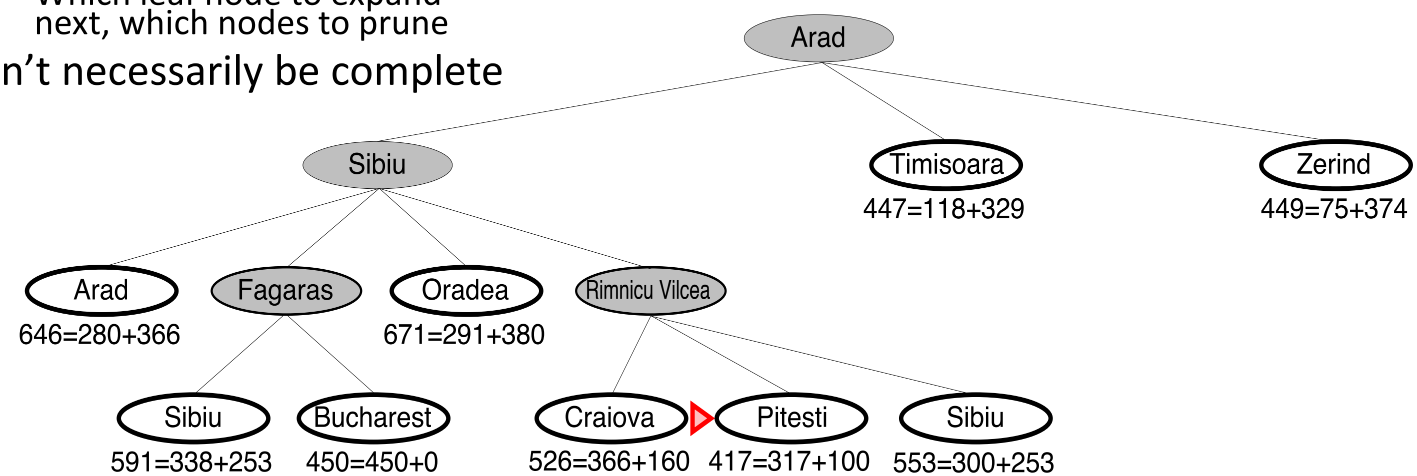
- Node: a pair $v = (\pi, s)$
  - $s = \gamma(s_0, \pi)$
  - In practice, v may contain other things
    - pointer to parent, $cost(\pi)$, …
  - π not always stored explicitly, can be computed from the parent pointers
- Children of $v = \{(\pi.a, \gamma(s,a)) \mid a$ is applicable in $s\}$
  - $\pi.a$: concatenation of π and a
- Successors of v
  - Children, children of children, etc.
- Ancestors of v
  - Nodes that have v as a successor
- Initial/starting node: $v_0 = (\langle\rangle, s_0)$
  - Root of the search tree
- Path in the search space
  - Sequence $\langle v_0, v_1, \ldots, v_n \rangle$ s.t. each v_i is a child of v_{i-1}
- Height of search space
  - length of longest acyclic path from v_0
- Depth of v
  - Length of path from v_0 to v, $length(\pi)$
- Branching factor of v
  - Number of children
- Branching factor of search tree
  - max branching factor of the nodes
- Expand v
  - Generate all children



Forward Search

- 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
  - Depends on how you implement the nondeterministic choice
    - Which leaf node to expand next, which nodes to prune
  - Won't necessarily be complete

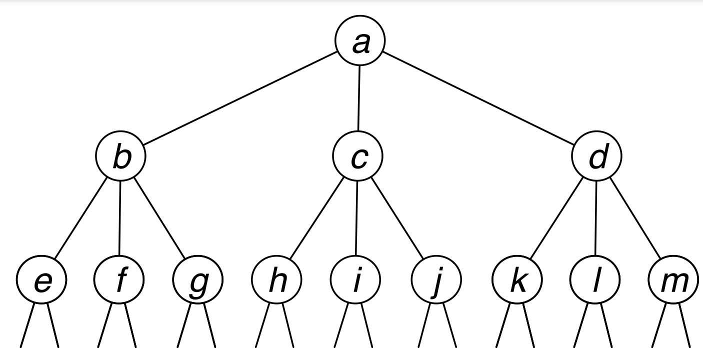
```
Forward-search(Σ,s0,g)
    s ← s0
    π ← ⟨⟩
    loop
        if s satisfies g then
            return π
        A' ←{a ∈ A | a is applicable in s}
        if A' = ∅ then
            return failure
        nondeterministically choose a ∈ A'
        s ← γ(s,a)
        π ← π.a
```



Deterministic Version

- Special cases
  - depth-first
  - breadth-first
  - A*
- Classify by
  - how they select nodes (step i)
  - how they prune nodes (step ii)
- Cycle checks during pruning:
  - Remove from children every node (π, s) that has an ancestor (π', s') s.t. $s' = s$
  - In classical planning, S is finite
    - Cycle-checking will guarantee termination

```
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
```

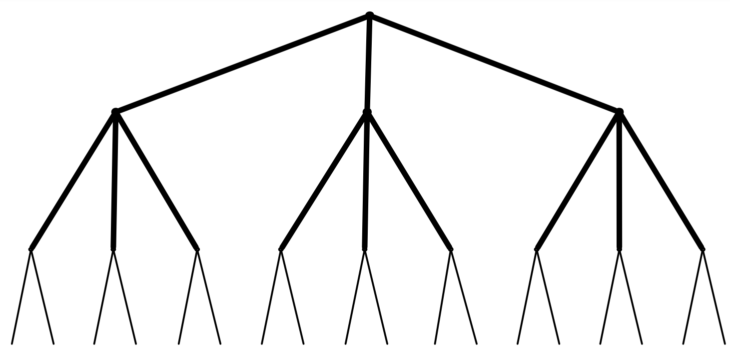


Breadth-first search (BFS)

- (i) select $(\pi, s) \in Frontier$ with smallest $length(\pi)$
  - tie-breaking rule: select oldest
- (ii) remove every $(\pi, s) \in Children \cup Frontier$ s.t. s is in $Expanded$
  - thus expand states at most once
- Properties
  - Terminates
  - Returns solution if one exists
    - Shortest, but not least-cost (except if shortest=least-cost)
  - Worst-case complexity:
    - memory $O(|S|)$
    - running time $O(b|S|)$

    where
    - b = max branching factor
    - $|S|$ = number of states in S

```
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
```

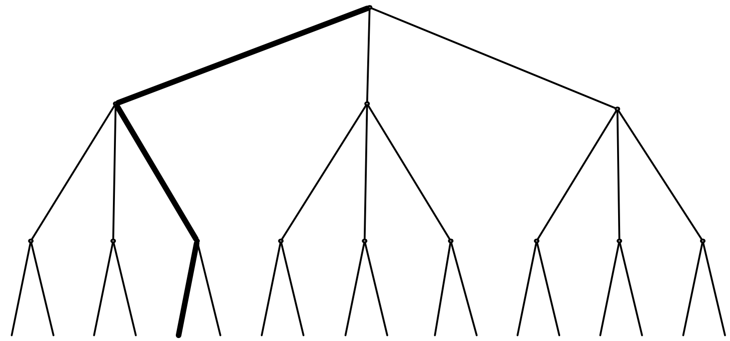


Depth-First Search (DFS)

- (i) Select $(\pi, s) \in \mathit{Children}$ that has largest $\mathit{length}(\pi)$
  - Possible tie-breaking rules: left-to-right, smallest $\mathit{height}(s)$
- (ii) do cycle-checking, then prune all nodes that recursive DFS would discard
  - Repeatedly remove from $\mathit{Expanded}$ any node that has no children in $\mathit{Children} \cup \mathit{Frontier} \cup \mathit{Expanded}$
- Properties
  - Terminates
  - Returns solution if there is one
    - No guarantees on quality
  - Worst-case complexity
    - Running time $O(b^l)$
    - Memory $O(bl)$

    Where
    - b = max branching factor
    - l = max depth of any node

```
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
```

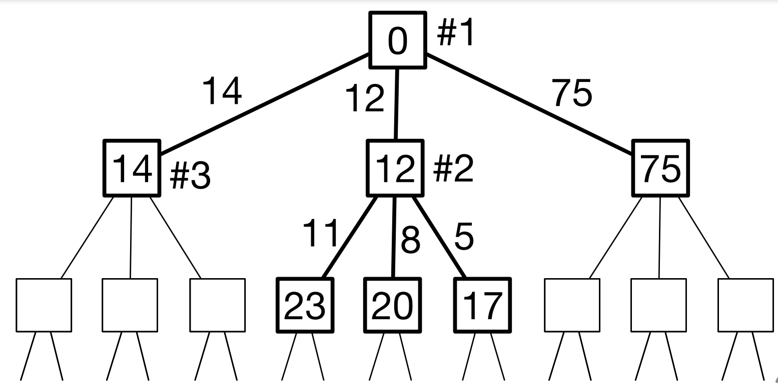


Uniform-Cost Search

- (i) Select $(\pi, s) \in Children$ that has smallest $cost(\pi)$
- (ii) Prune every $(\pi, s) \in Children \cup Frontier$ such that $Expanded$ already contains a node (π', s)
  - $cost(\pi') \leq cost(\pi)$, so we only keep the least-cost path to s
- Properties
  - Terminates
  - Finds optimal solution if one exists
  - Worst-case complexity
    - Time $O(b|S|)$
    - Memory $O(|S|)$

    where
    - b = max branching factor
    - $|S|$ = number of states in S

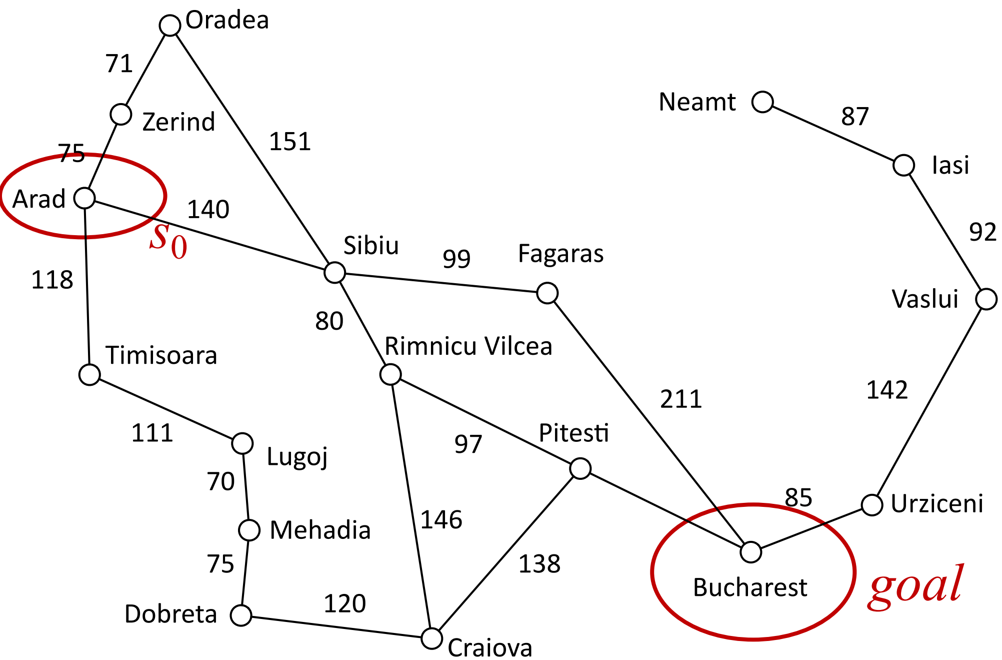
```
Deterministic-Search(Σ, s₀, g)
    Frontier ← {(⟨⟩, s₀)}
    Expanded ← ∅
    while Frontier ≠ ∅ do
        select a node ν = (π, s) ∈ Frontier   (i)
        remove ν from Frontier
        add ν to Expanded
        if s satisfies g then
            return π
        Children ←
            {(π.a, γ(s,a))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



Heuristic Function

- Motivation: get to a solution quickly by selecting nodes close to the goal
  - Compare: A*
- Let $h^*(s) = \min\{cost(\pi) \mid \gamma(s, \pi) \text{ satisfies } g\}$
  - Note that $h^*(s) \geq 0$ for all s
- Heuristic function $h(s)$:
  - Returns an estimate of $h^*(s)$
    - Assume $h(s) \geq 0$ for all s
  - Properties
    - h is admissible if for every s, $h(s) \leq h^*(s)$
    - h is ε-admissible if for every s, $h(s) \leq h^*(s) + \varepsilon$
- Let $v = (\pi, s)$ be a node
  - $f^*(v) = cost(\pi) + h^*(s)$
    - Min cost of all paths to goal that start with π
  - $f(v) = cost(\pi) + h(s)$
    - Estimate of $f^*(v)$

Example



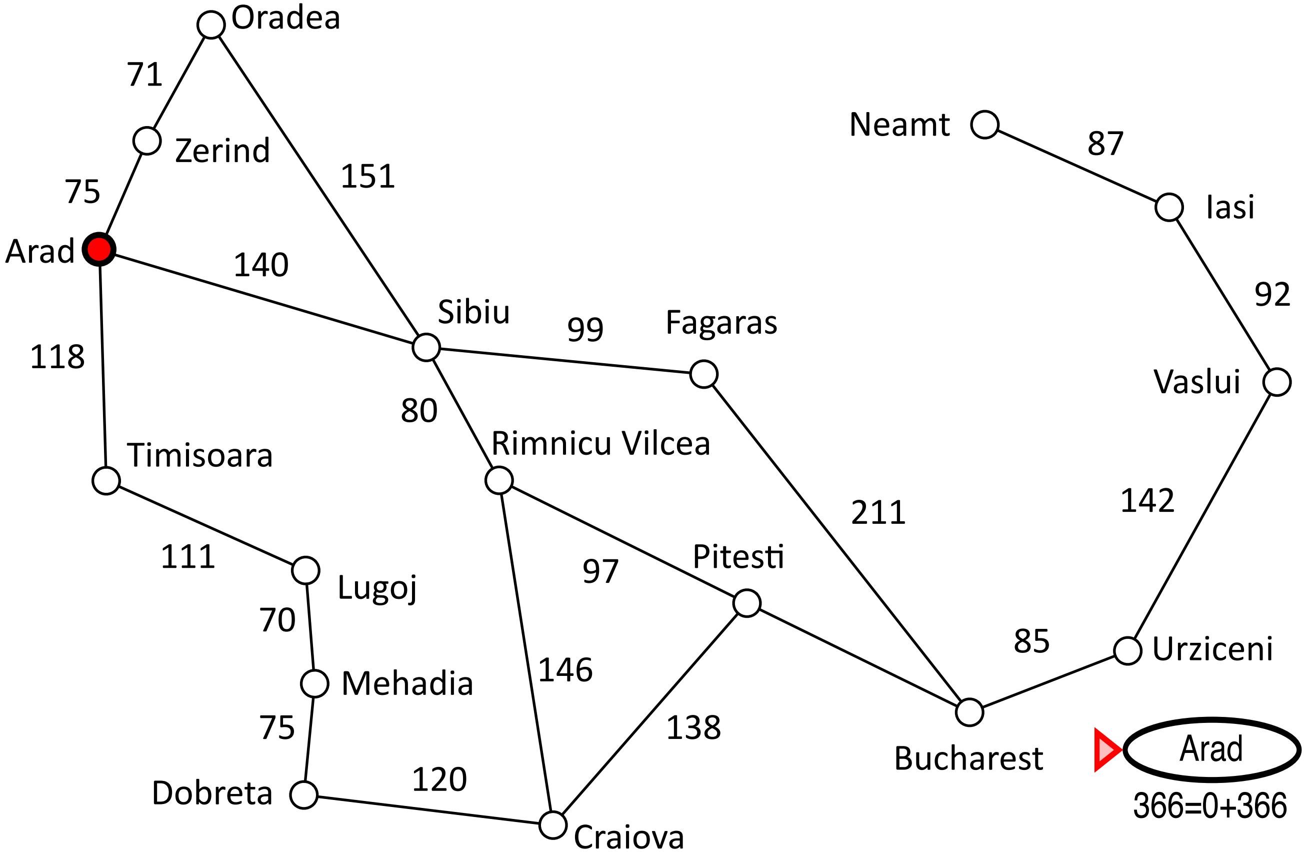
straight-line distance from s to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Fagaras	176
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

- State s = what city you are in
- Action: follow road from s to a neighboring city
- $h^*(s)$ = length of shortest sequence of roads from s to Bucharest
- $h(s)$ = straight-line distance from s to Bucharest
  - *domain-specific*; later we will discuss *domain-independent*
- $f^*\big((\pi, s)\big)$ = length of (π + shortest sequence of roads from s to Bucharest)

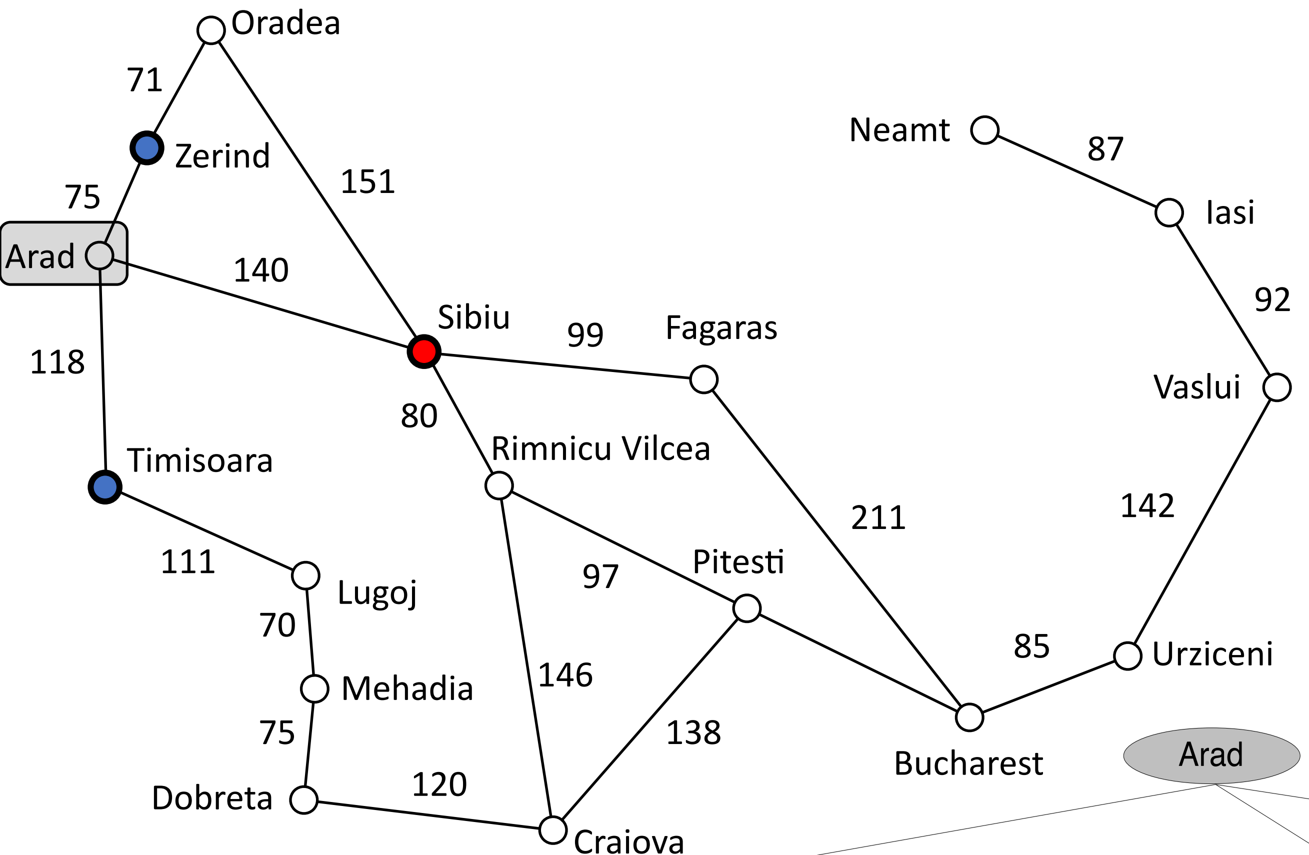
A*

- (i) Select a node $\nu = (\pi, s)$ in $Frontier$ that has smallest value of $f(\nu) = cost(\pi) + h(s)$
  - Tie-breaking rule: choose oldest
- (ii) for every node $\nu = (\pi, s)$ in $Children$
  - if $Children \cup Frontier \cup Expanded$ contains more than one node for s
    - then it has multiple paths to s
    - Keep only the one with the lowest f-value
  - Tie-breaking rule: keep oldest
- Properties
  - After upcoming example

```
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
```

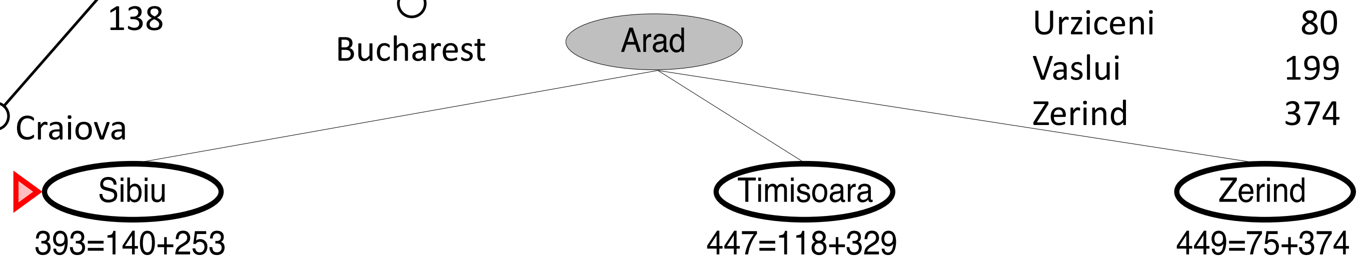


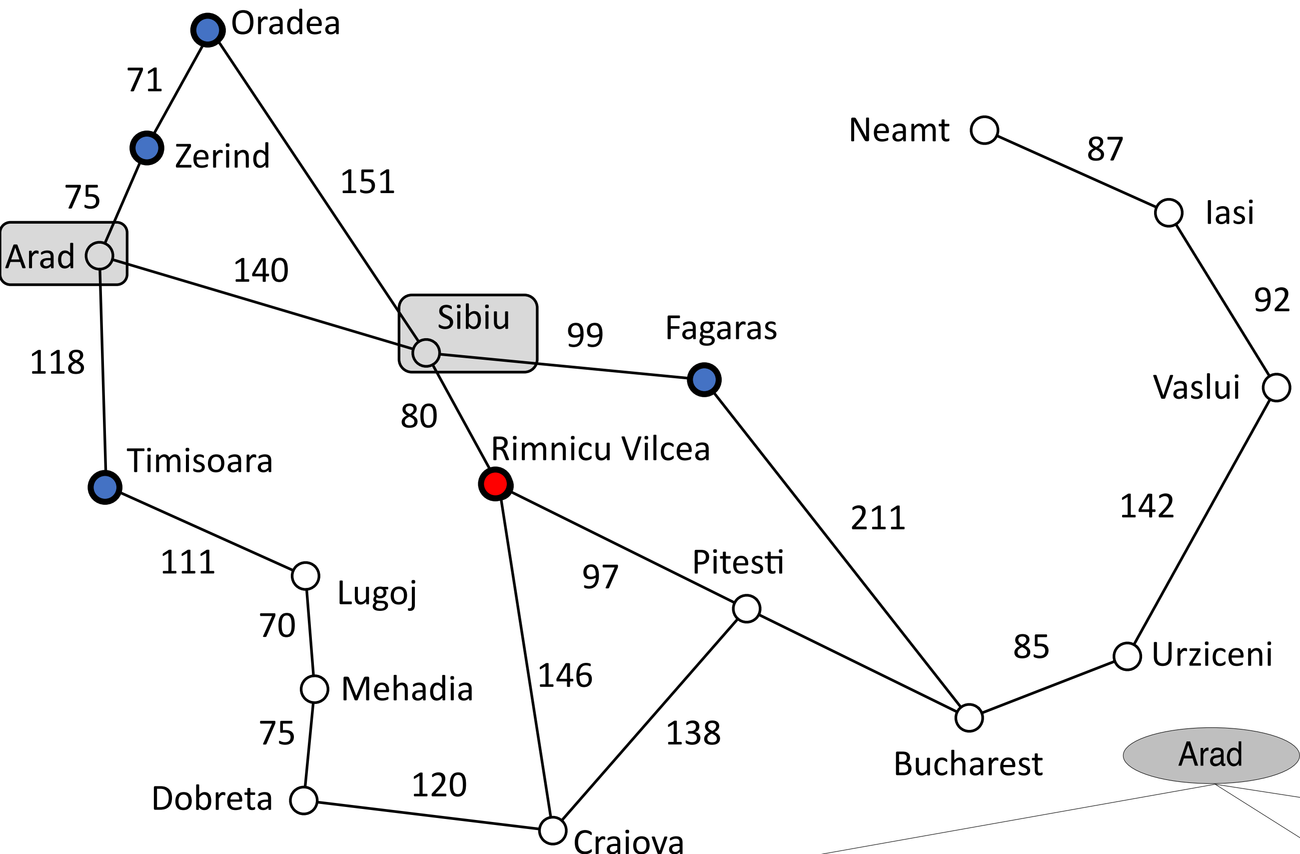
straight-line dist. from s to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Fagaras	176
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



straight-line dist.
from s to Bucharest

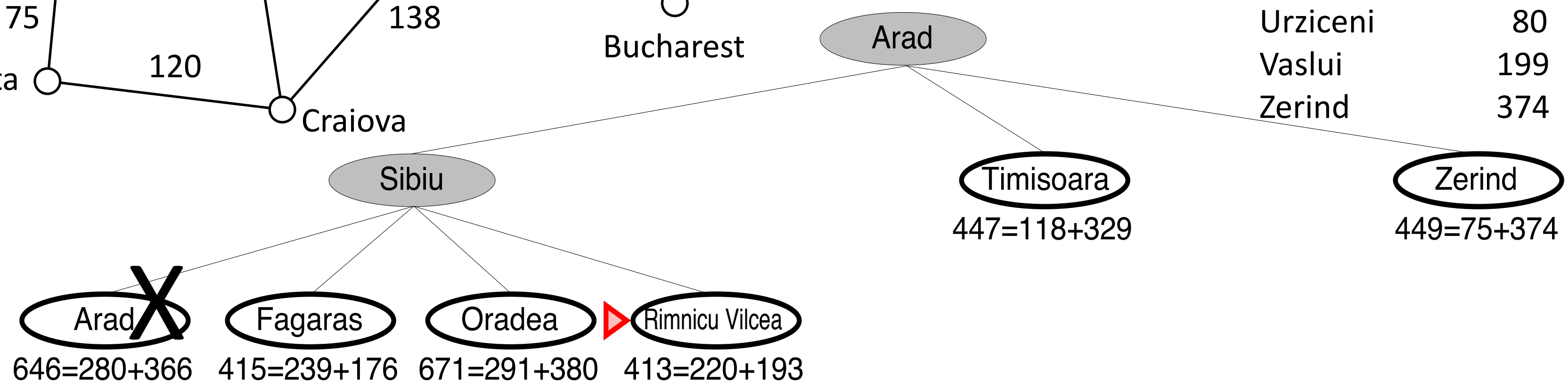
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Fagaras	176
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

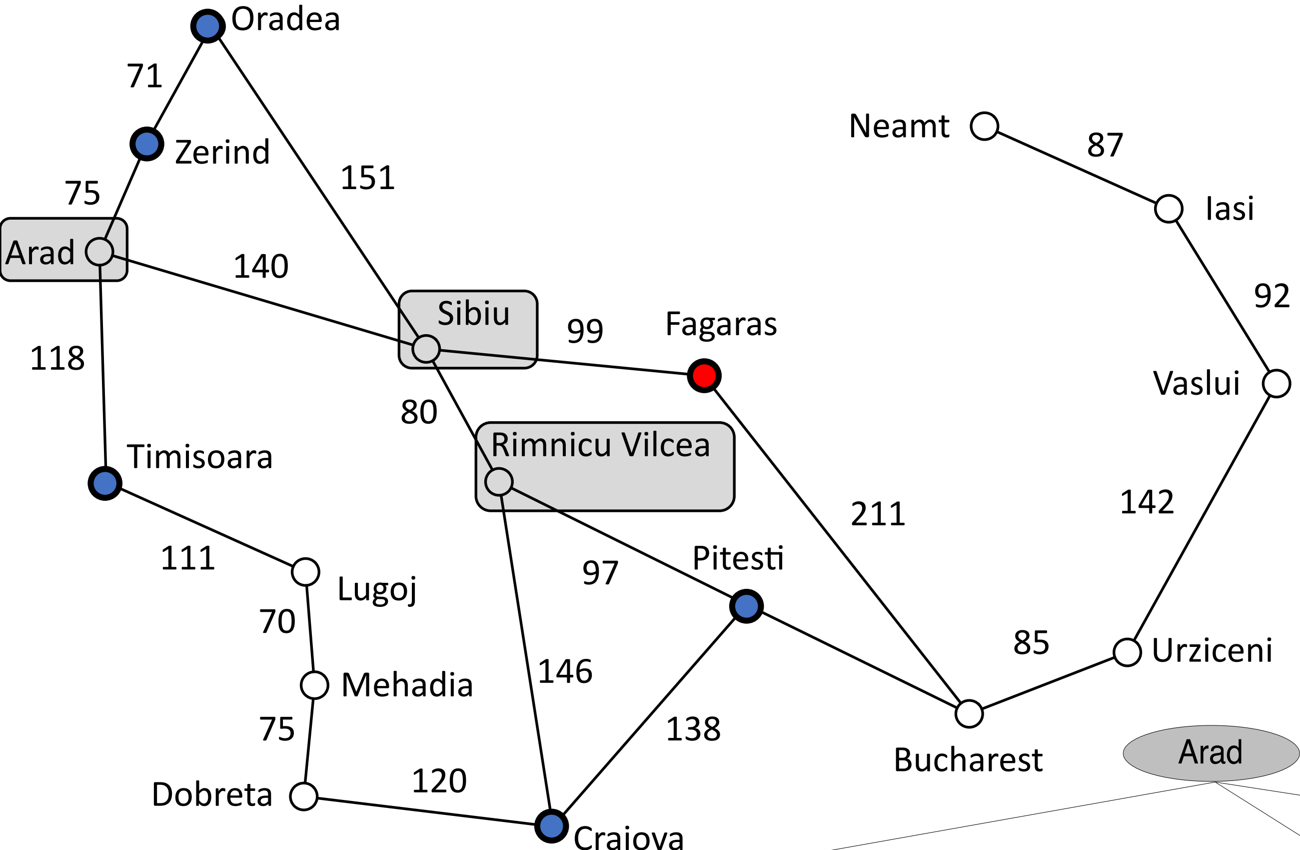




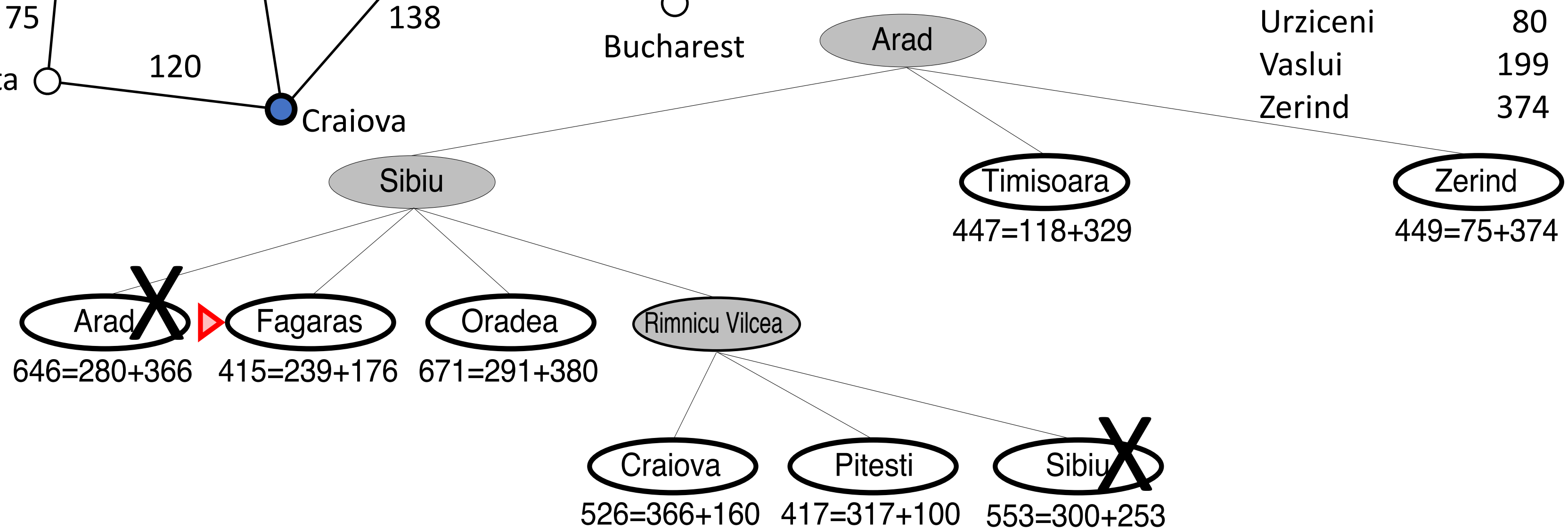
straight-line dist.
from s to Bucharest

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



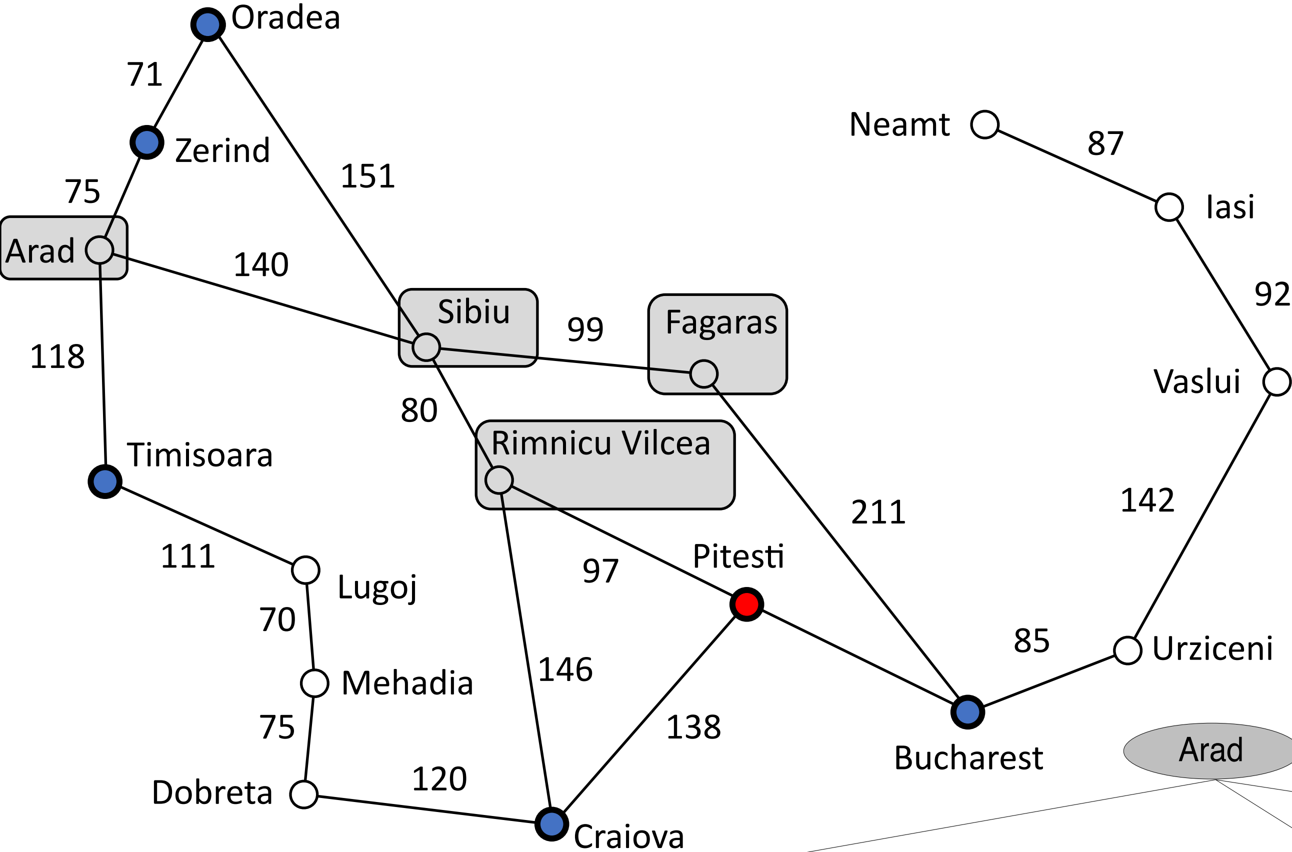


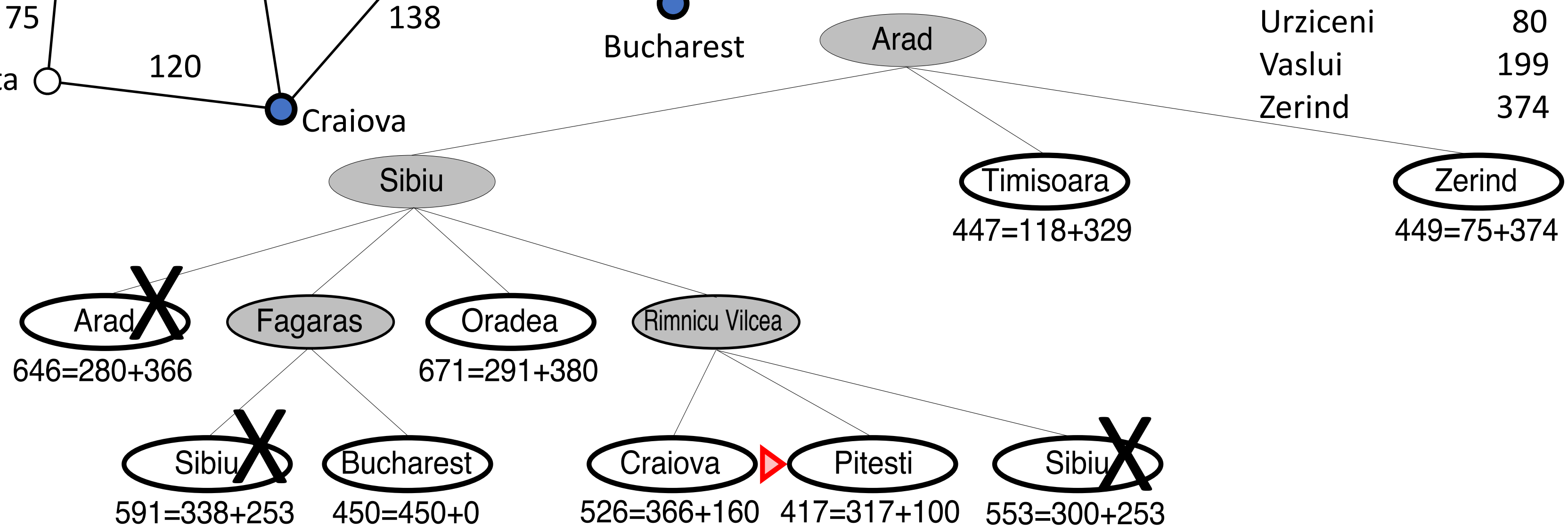
straight-line dist. from s to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Fagaras	176
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

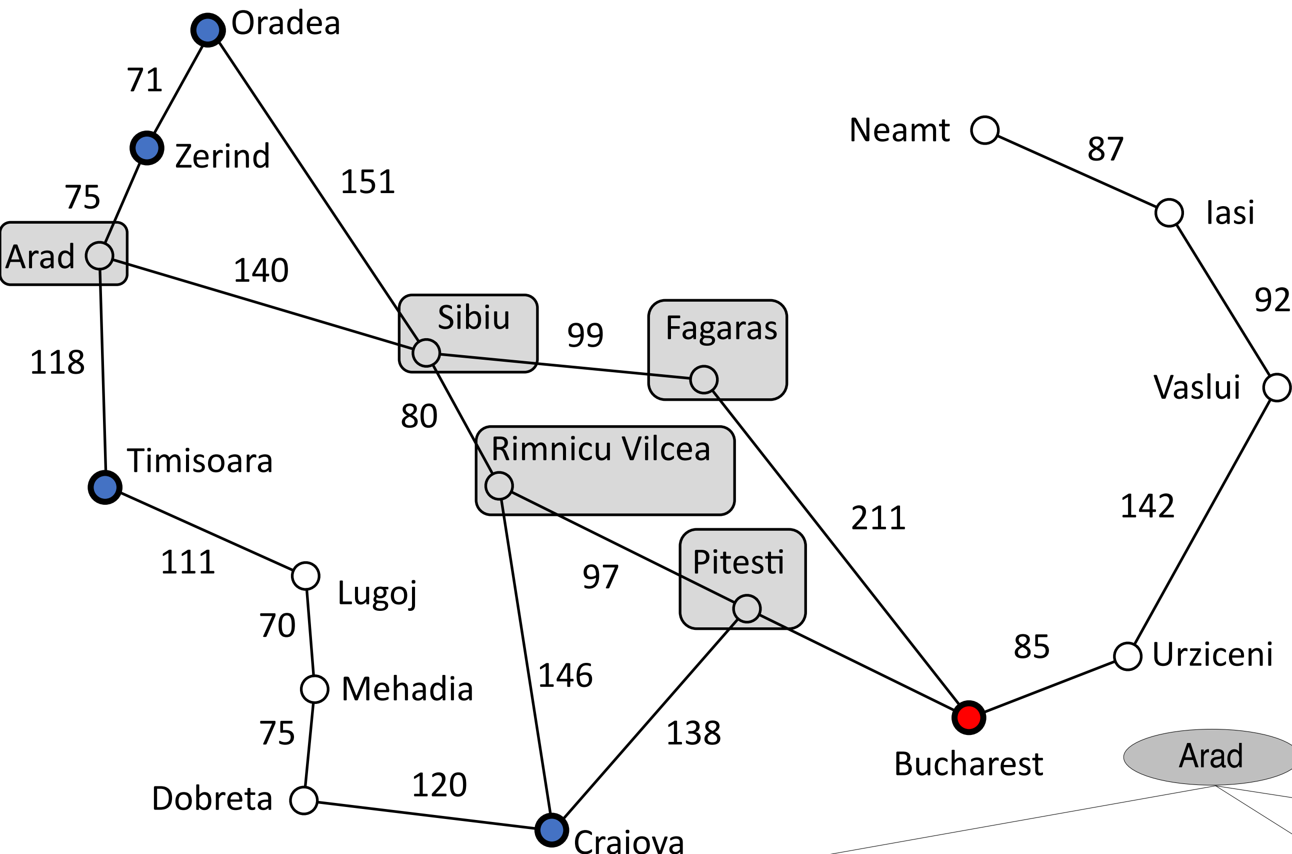


straight-line dist.
from s to Bucharest

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

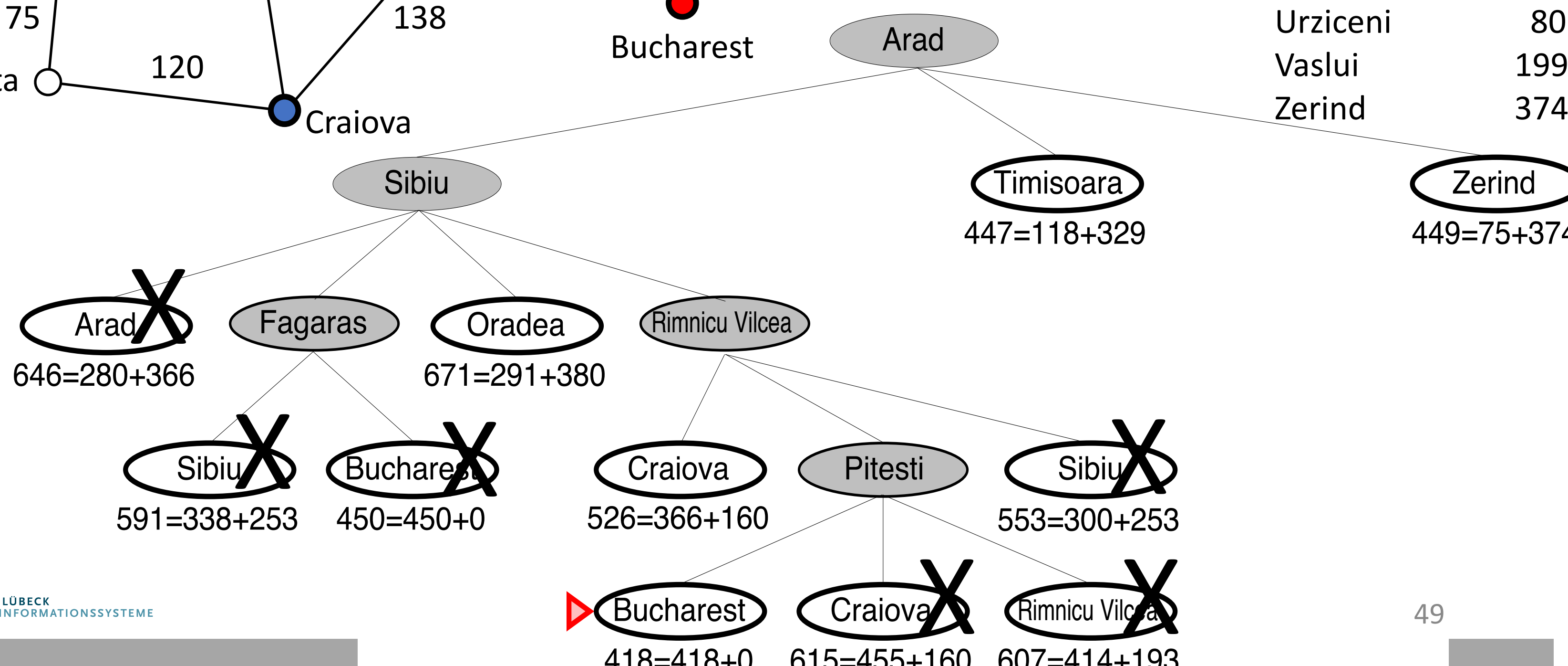






straight-line dist.
from s to Bucharest

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



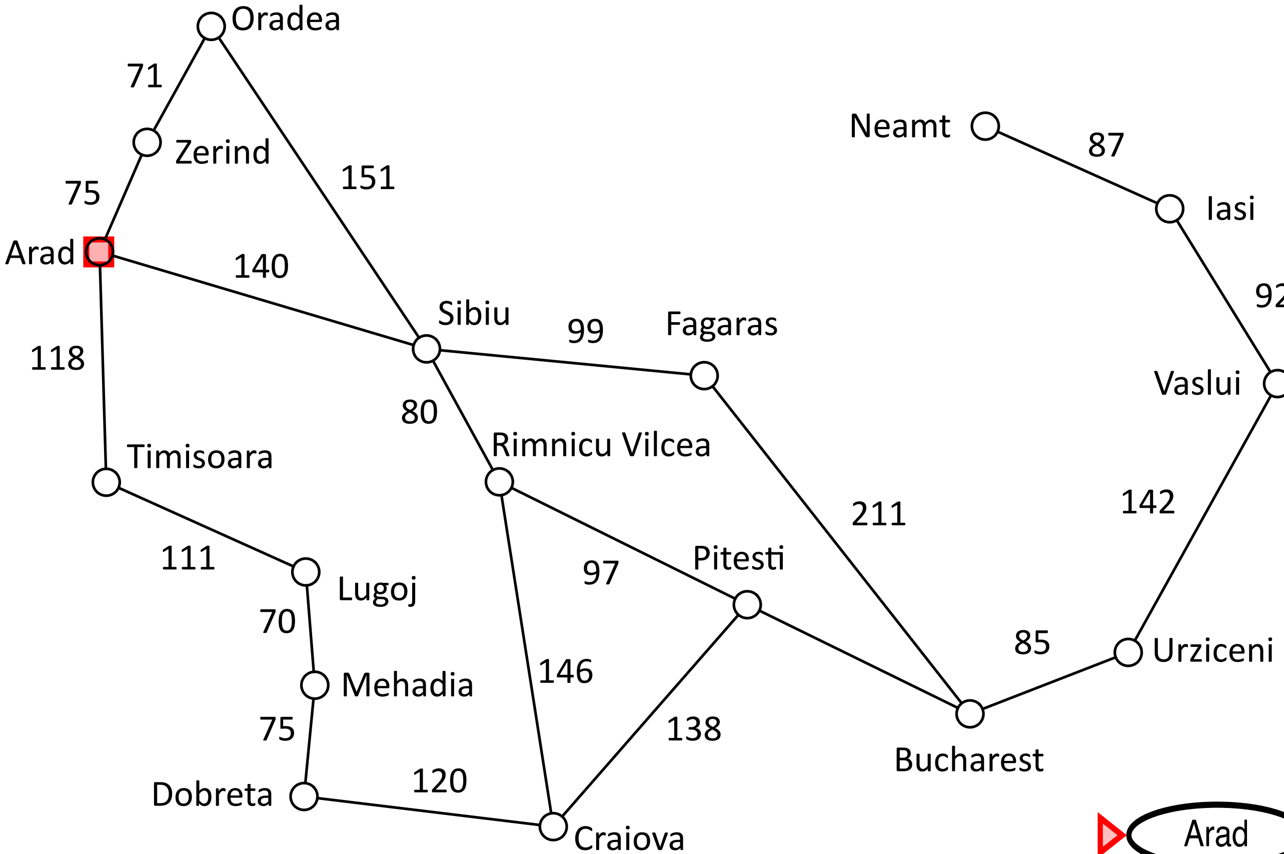
Properties of A*

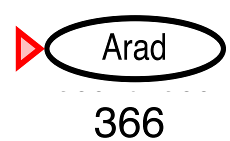
- In classical planning problems, A* will always terminate
- *Completeness:* if the problem is solvable, A* will return a solution
  - If h is admissible, then the solution will be optimal (least cost)
  - If h is ε-admissible, then the solution will be ε-optimal
- If h is monotone then
  - $f(\nu) \leq f(\nu')$ for every child ν' of a node ν
  - A* will expand nodes in non-decreasing order of f values
  - A* will never prune any nodes from $Expanded$
  - A* will expand no state more than once
- Definition: h dominates h' if $h'(s) \leq h(s) \leq h^*(s)$ for every s
  - If h dominates h' then (assuming ties are always resolved in favor of the same node)
    - A* will never expand more nodes with h than with h'
    - In most cases A* will expand fewer nodes with h than with h'
- A* needs to store every node it visits
  - Running time and memory both $O(b|S|)$ in worst case
  - With good heuristic function, usually much smaller

Greedy Best-First Search (GBFS)

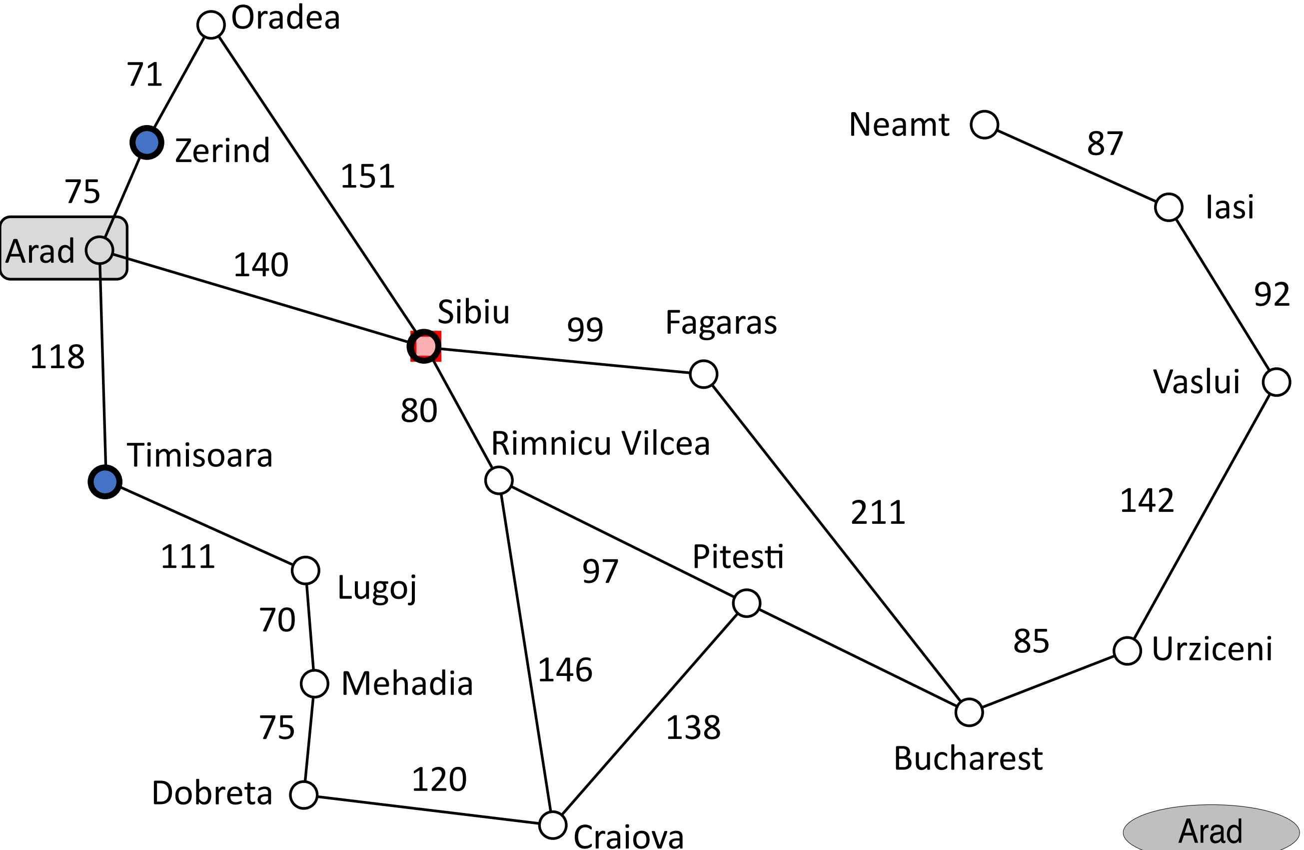
- Find a solution as quickly as possible, even if it isn't optimal
  - Select nodes that are likely to be on the least-cost path from where you are now
- (i) Select a node $(\pi, s) \in Frontier$ that has smallest $h(s)$
- (ii) same as in A*: for every node $\nu = (\pi, s)$ in $Children$
  - if $Children \cup Frontier \cup Expanded$ contains more than one node for s
    - then it has multiple paths to s
    - Keep only the one with the lowest f-value
  - Tie-breaking rule: keep oldest
- Properties
  - Terminates
  - Returns a solution if one exists
    - Often near-optimal
    - will usually find it quickly

```
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
```



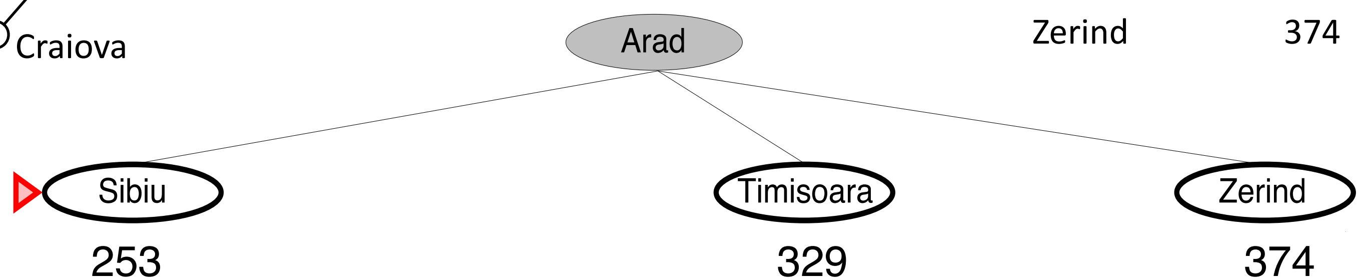


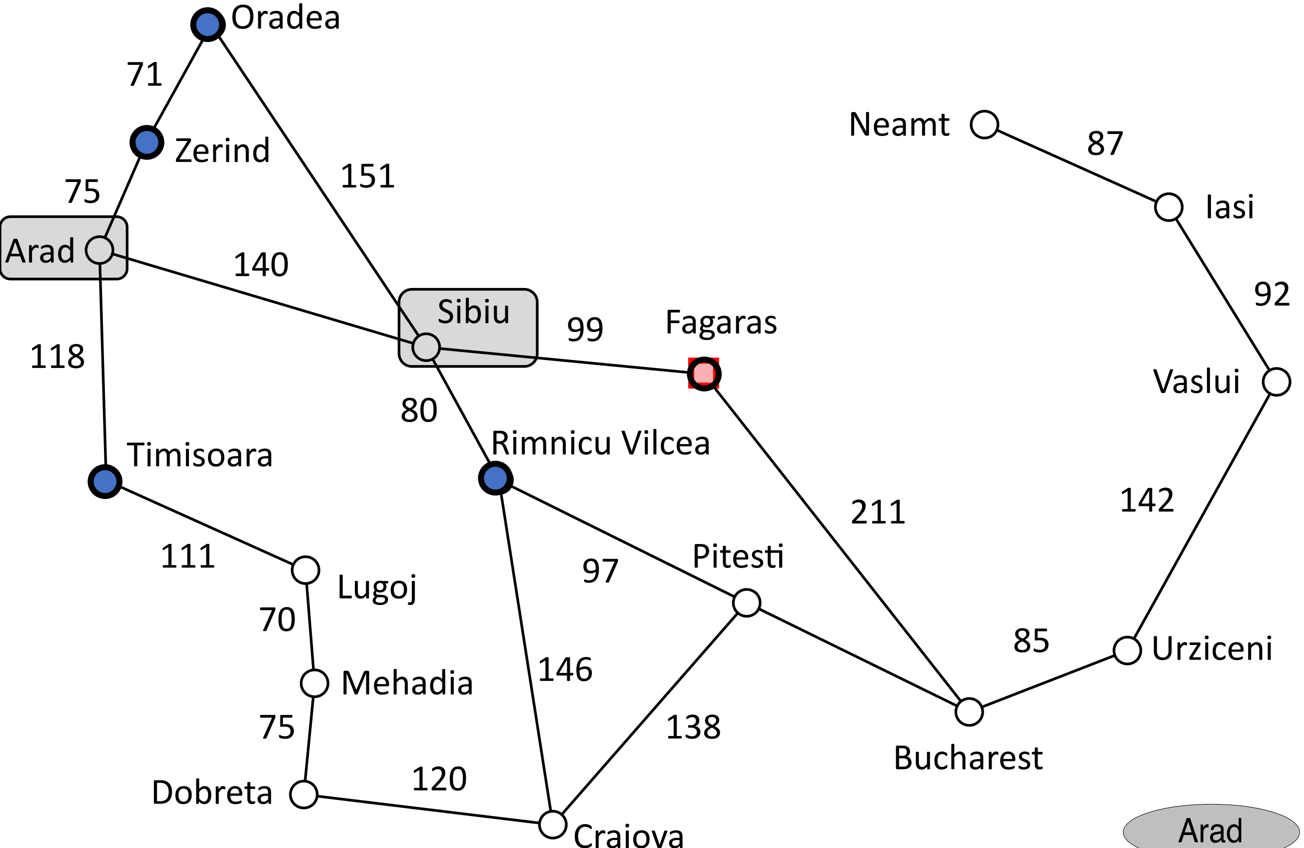
straight-line dist. from s to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Fagaras	176
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



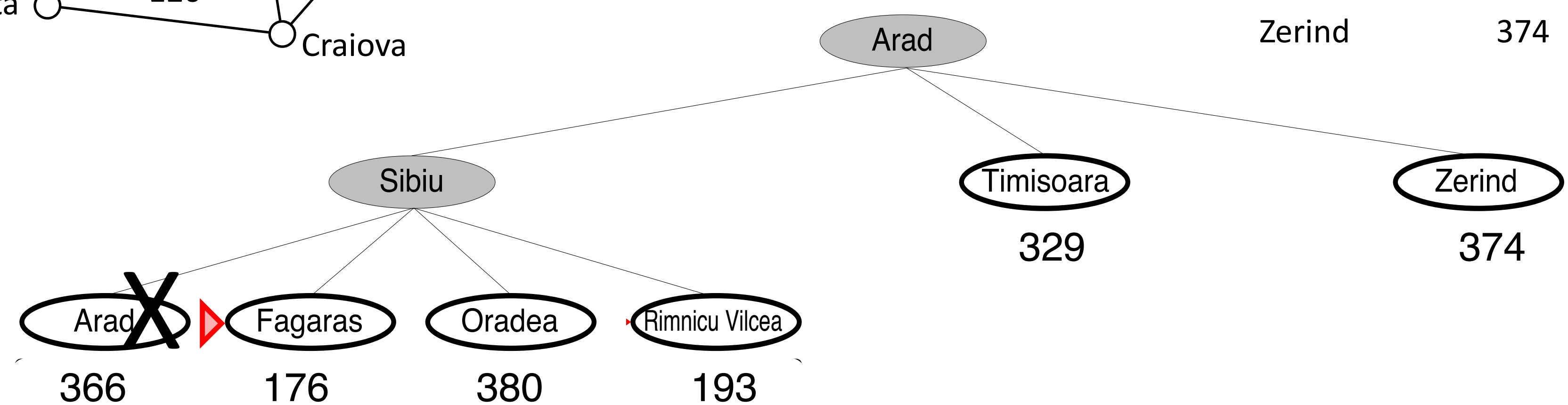
straight-line dist.
from s to Bucharest

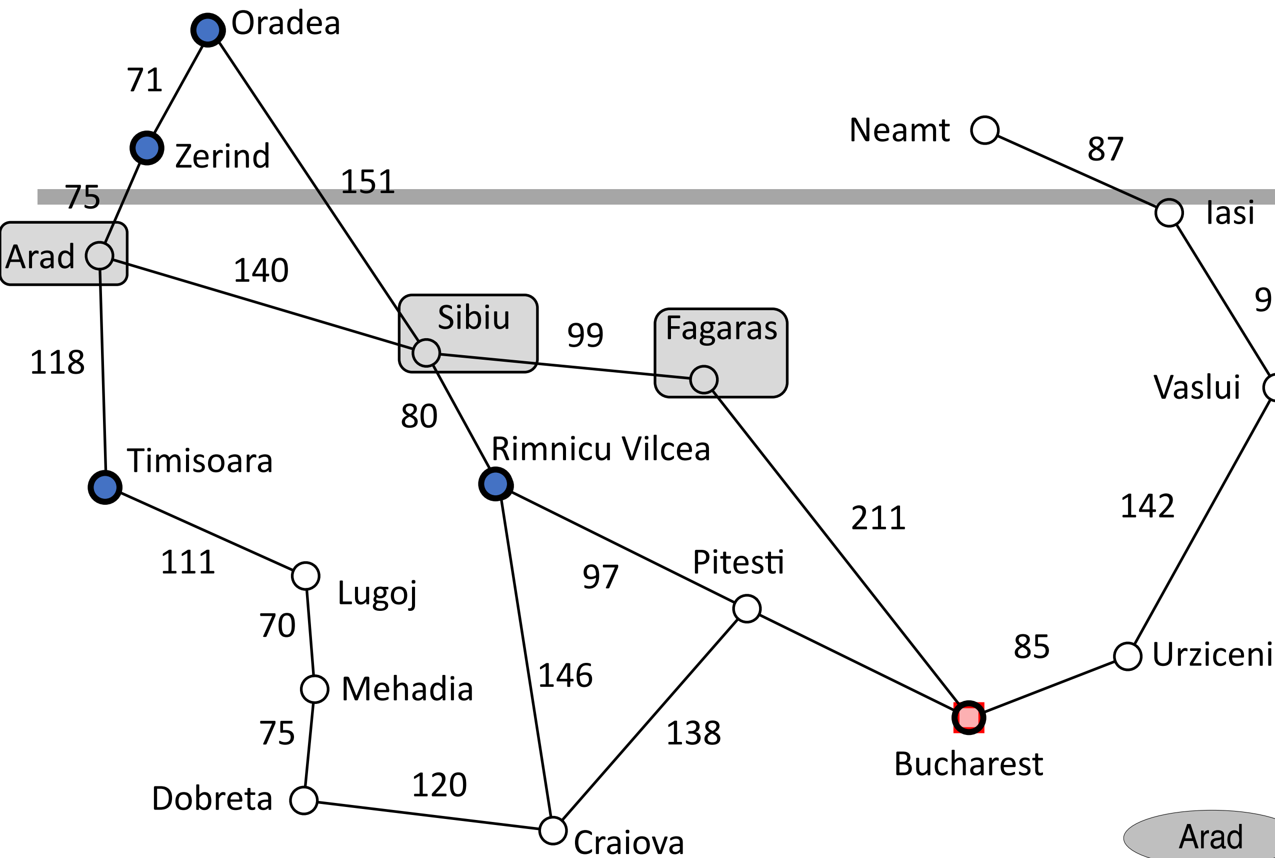
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Fagaras	176
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



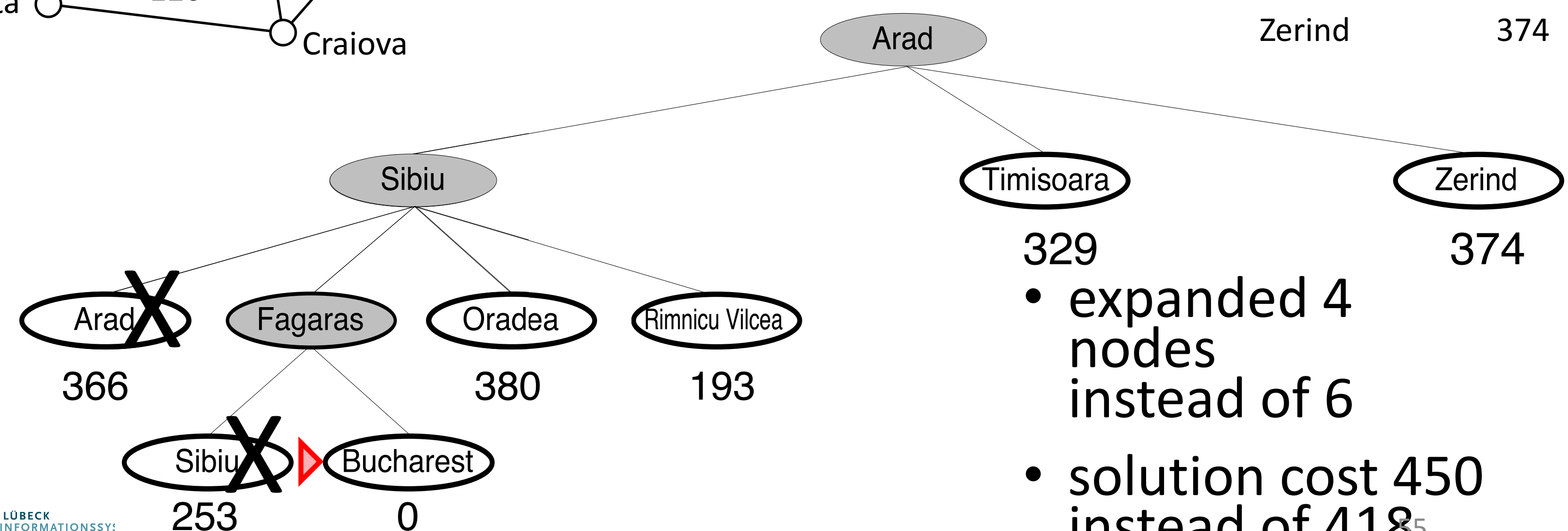


straight-line dist. from s to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Fagaras	176
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374





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



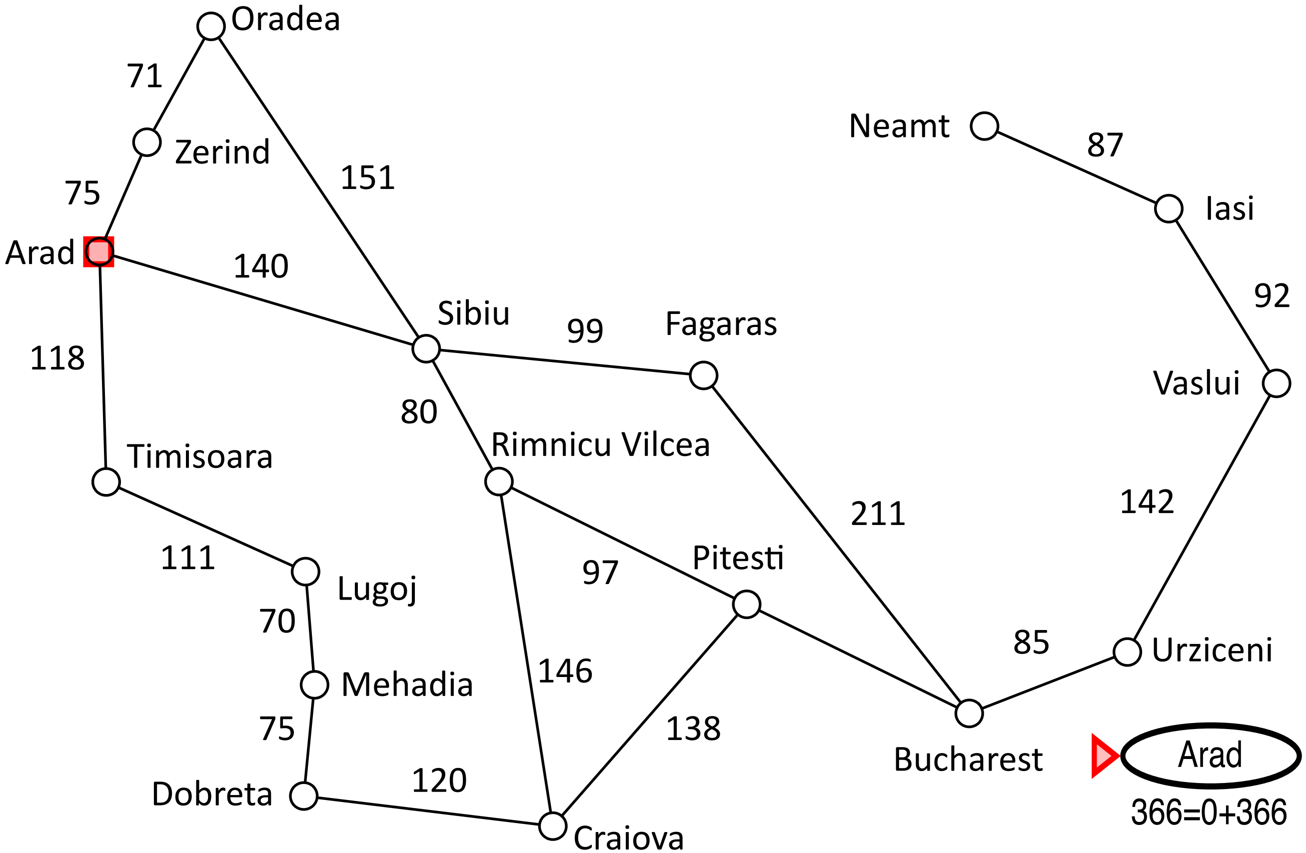
- expanded 4 nodes instead of 6
- solution cost 450 instead of 418

UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR INFORMATIONSSYSTEME

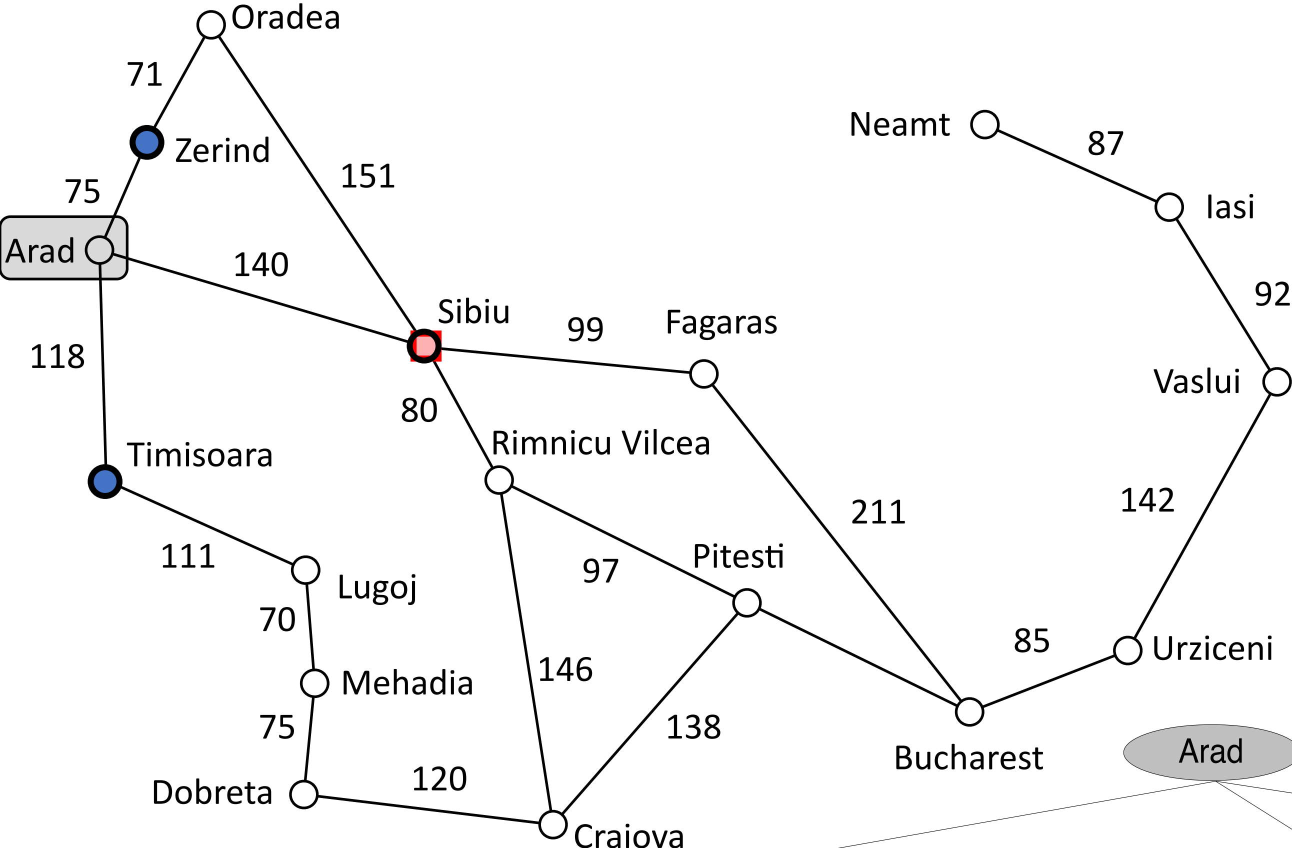
Depth-First Branch and Bound (DFBB)

- (i) same as DFS
  - Select $\nu = (\pi, s) \in Children$ that has largest $length(\pi)$
  - Tie-breaking: smallest $height(s)$
- (ii) Prune
  - Like DFS
    - do cycle-checking and prune what recursive DFS would discard
  - Additional pruning during node expansion:
    - If $f(\nu) \geq c^*$, then discard ν
- Properties
  - Termination, completeness, optimality same as A*
  - Usually less memory than A*, but more time
  - Worst-case like DFS:
    - $O(bl)$ memory
    - $O(bl)$ running time

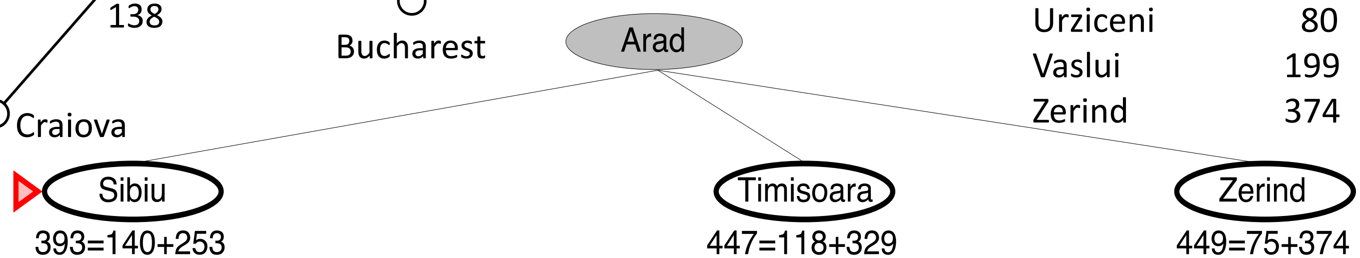
```
Deterministic-Search(Σ,s₀,g)
    Frontier ← {(⟨⟩, s₀)}
    Expanded ← ∅
    c*← ∞
    π*← failure
    while Frontier ≠ ∅ do
        select a node ν = (π,s) ∈ Frontier  (i)
        remove ν from Frontier
        add ν to Expanded
        ~~if s satisfies g then~~
            ~~return π~~
        if s satisfies g and cost(π) < c* then
            c*← cost(π); π*← π
        else if f(ν) < c* then
            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~~ π*
```

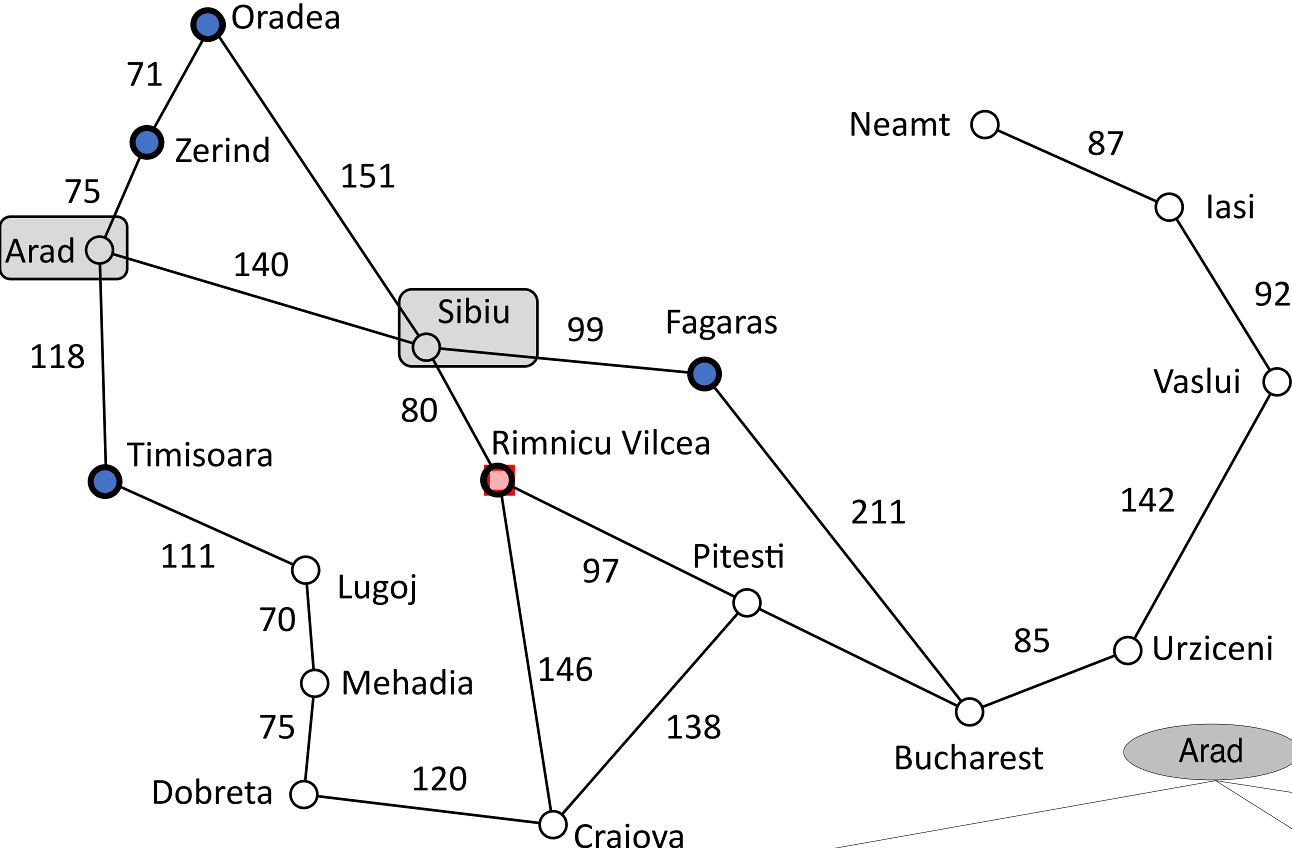


straight-line dist. from s to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Fagaras	176
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

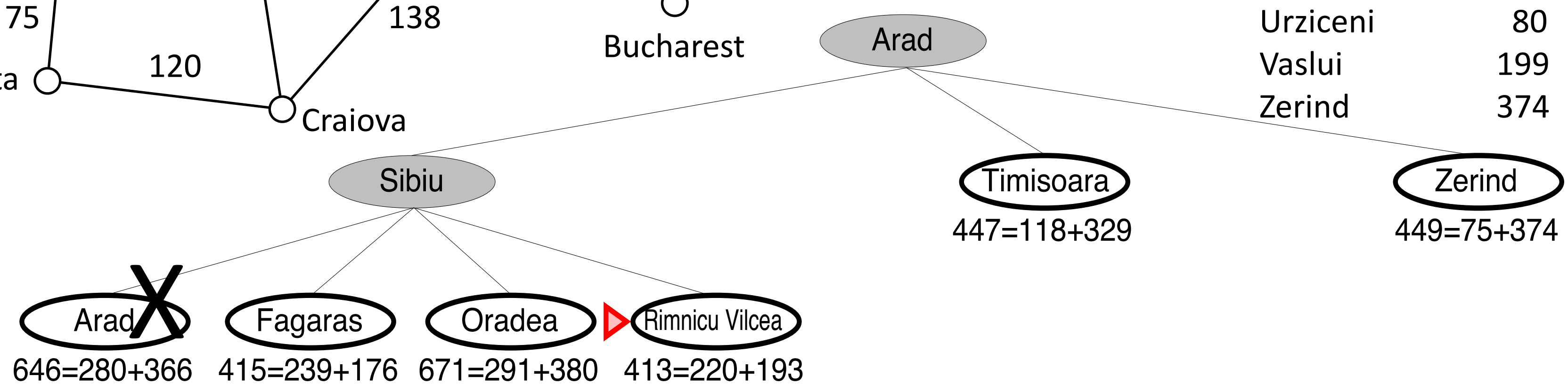


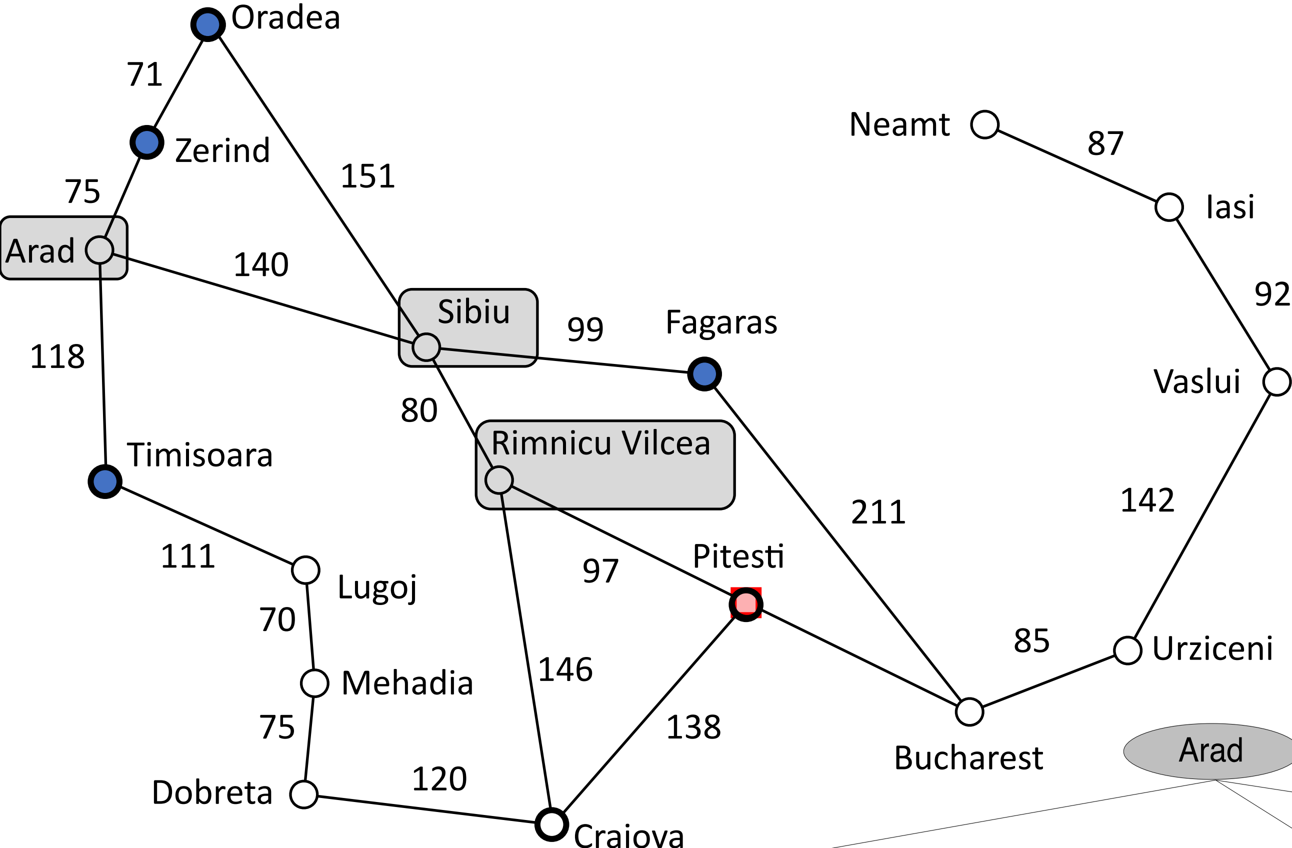
straight-line dist. from s to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Fagaras	176
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



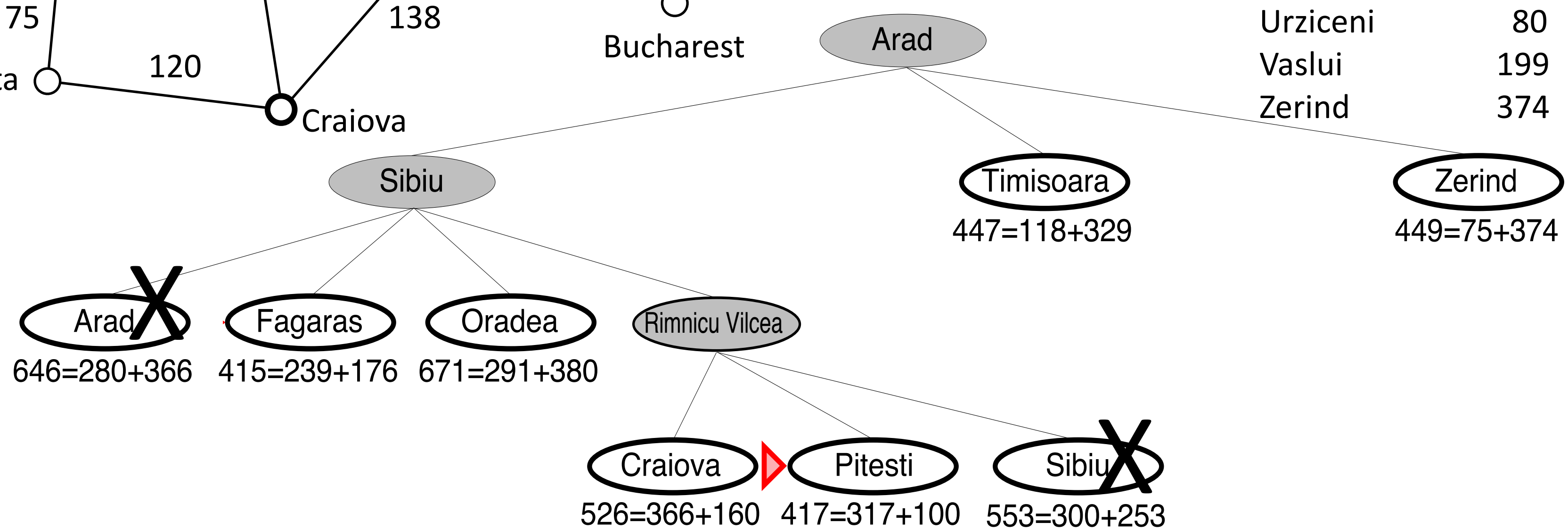


straight-line dist. from s to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Fagaras	176
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

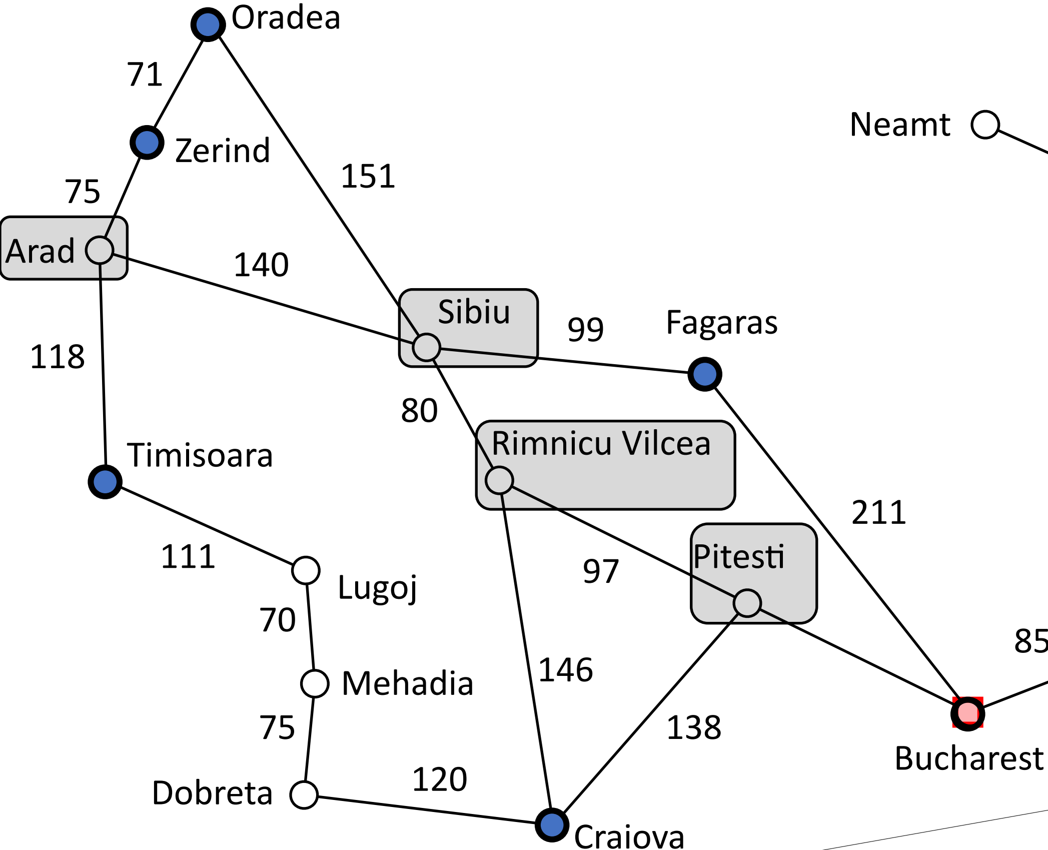


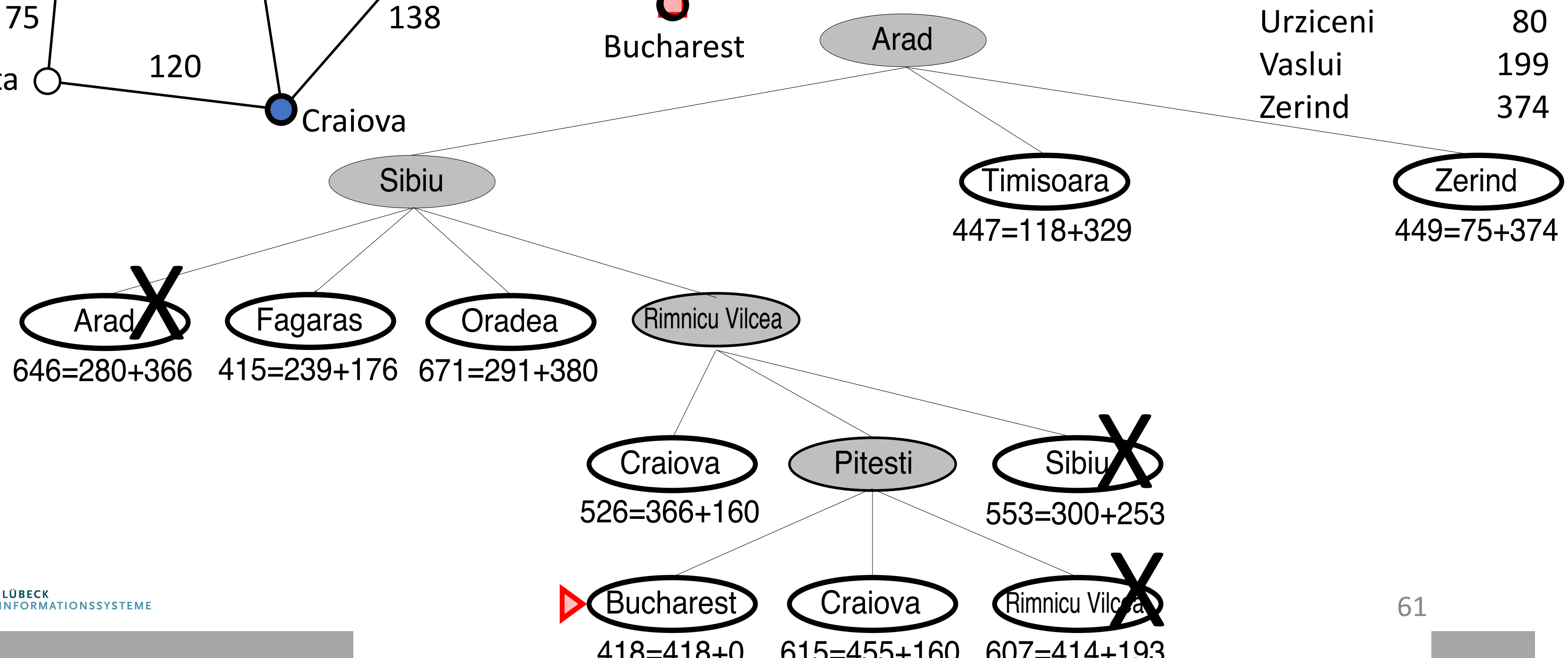


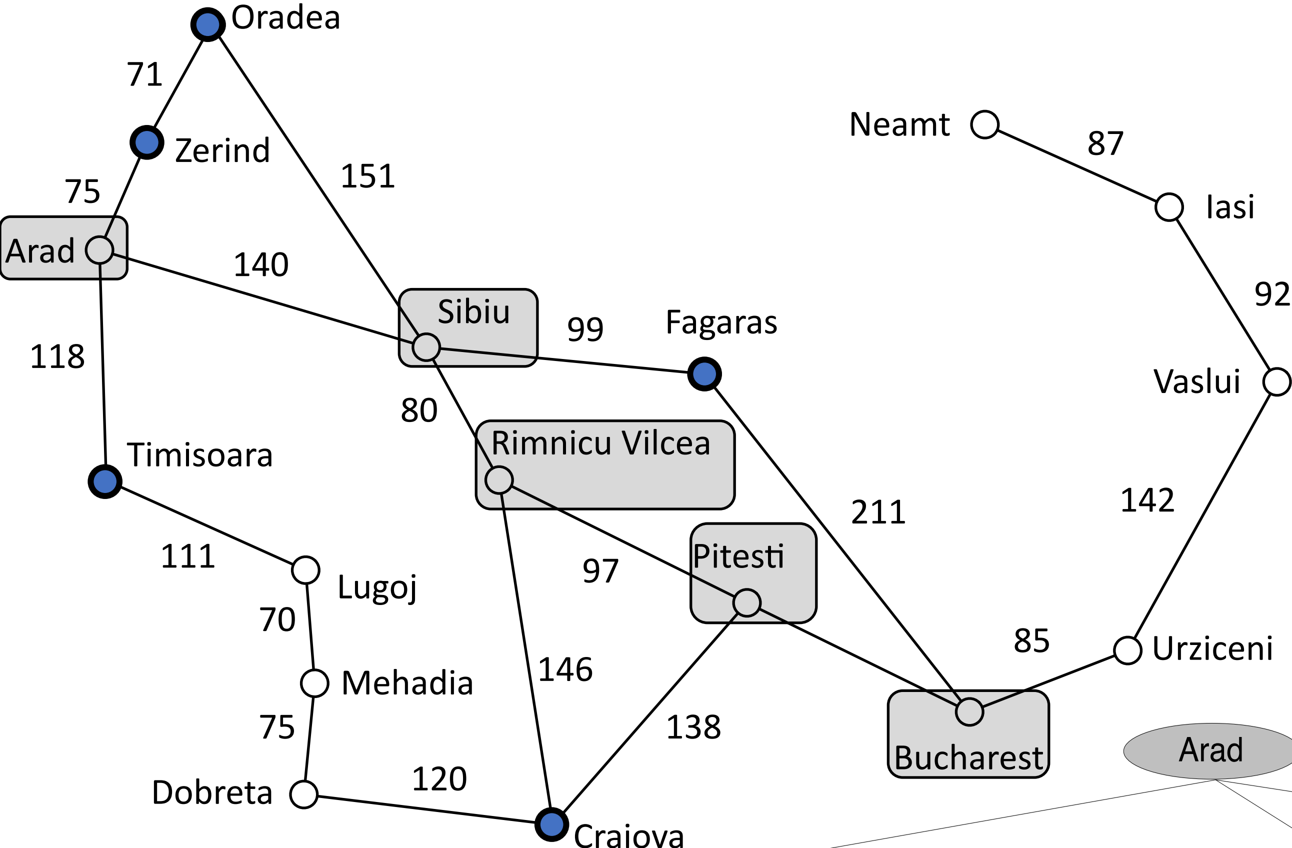
straight-line dist. from s to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Fagaras	176
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



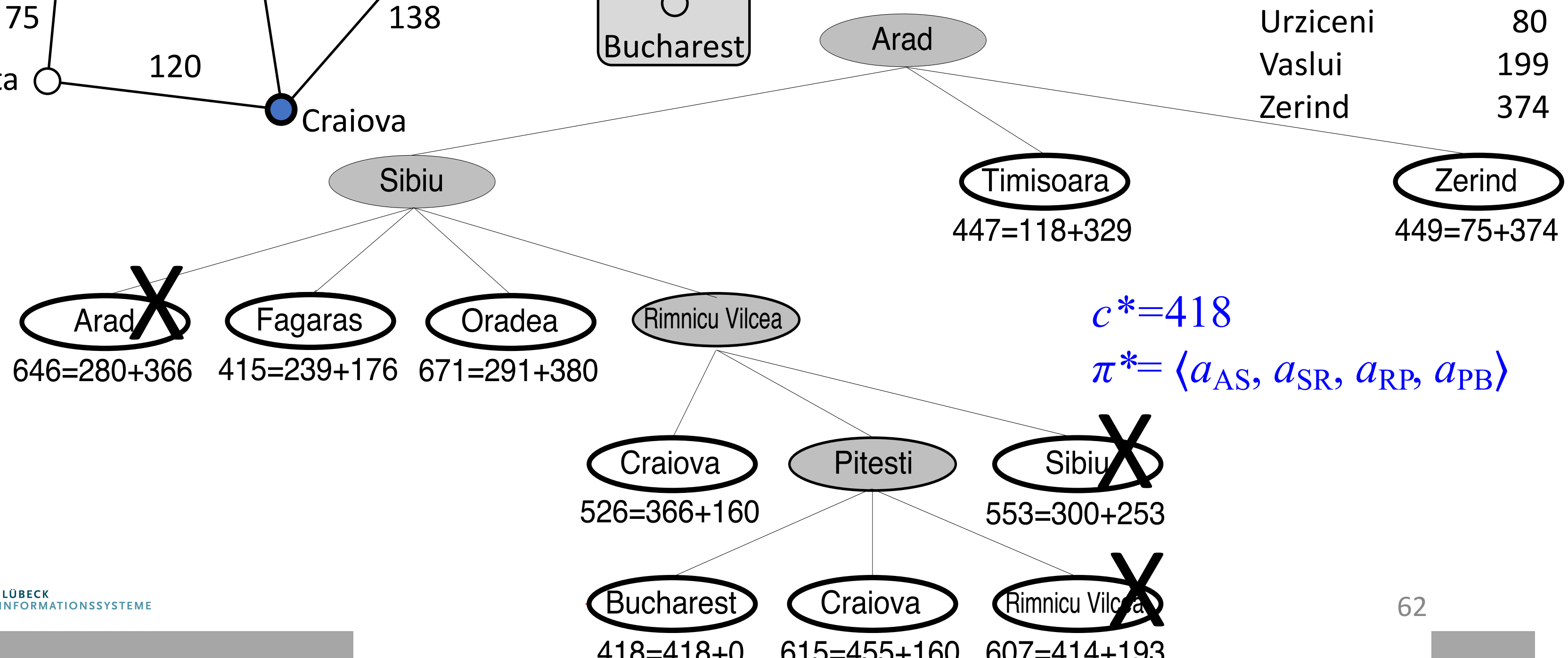
straight-line dist. from s to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Fagaras	176
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374





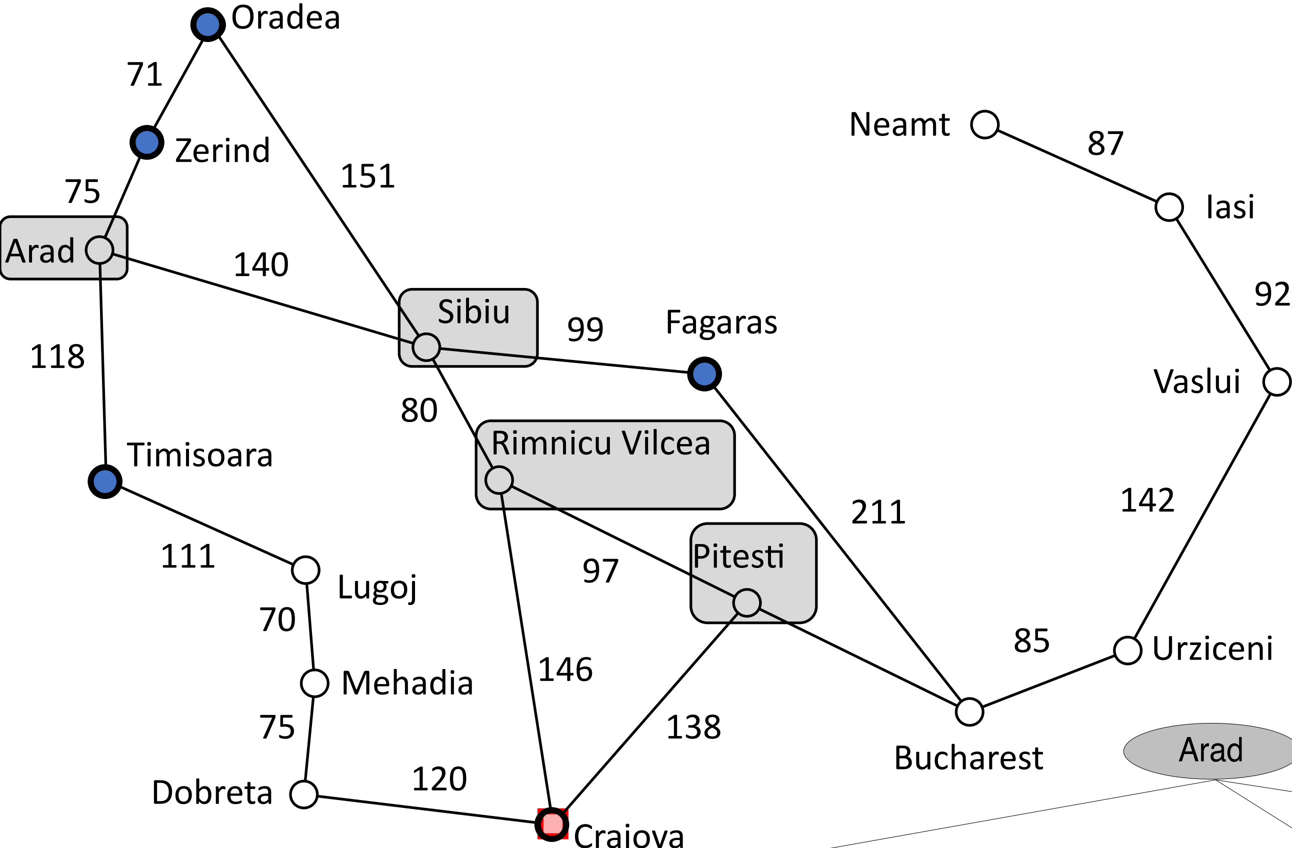


straight-line dist. from s to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Fagaras	176
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

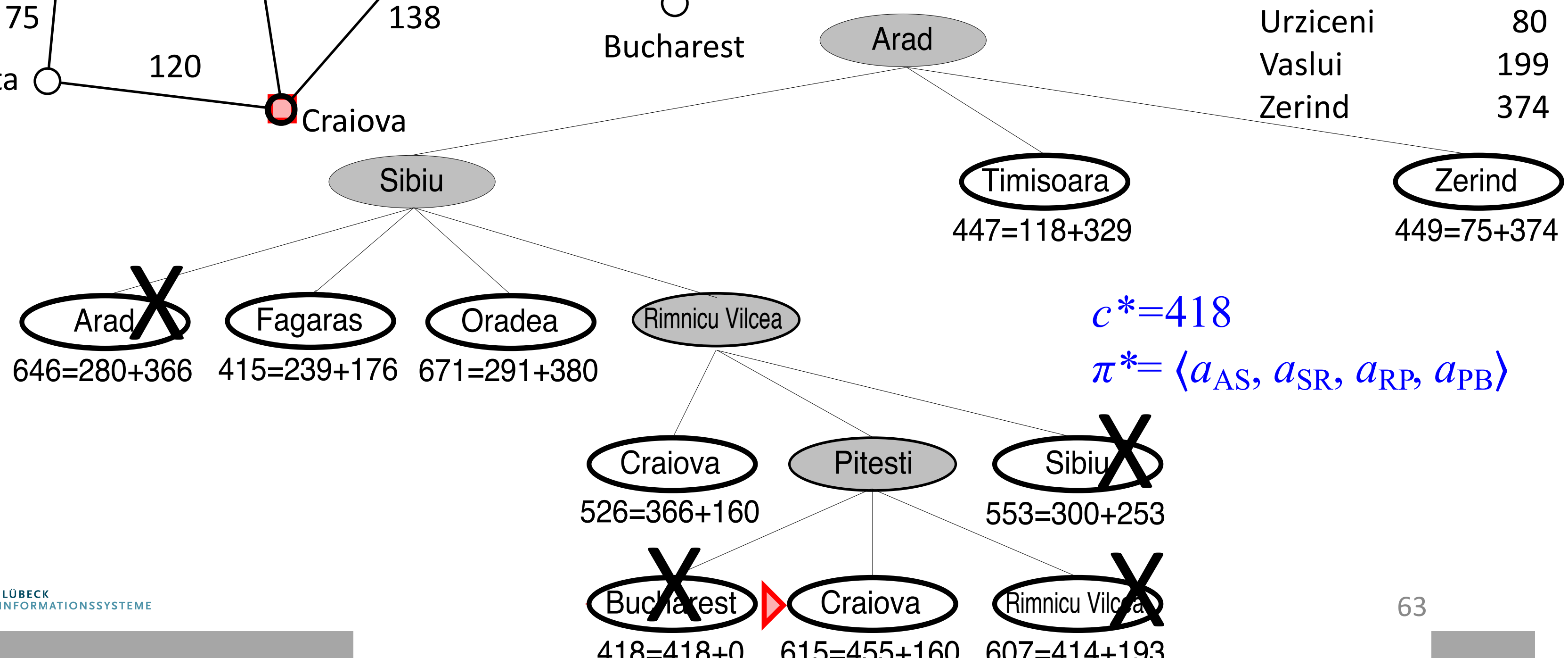


$c^* = 418$

$\pi^* = \langle a_{AS}, a_{SR}, a_{RP}, a_{PB} \rangle$

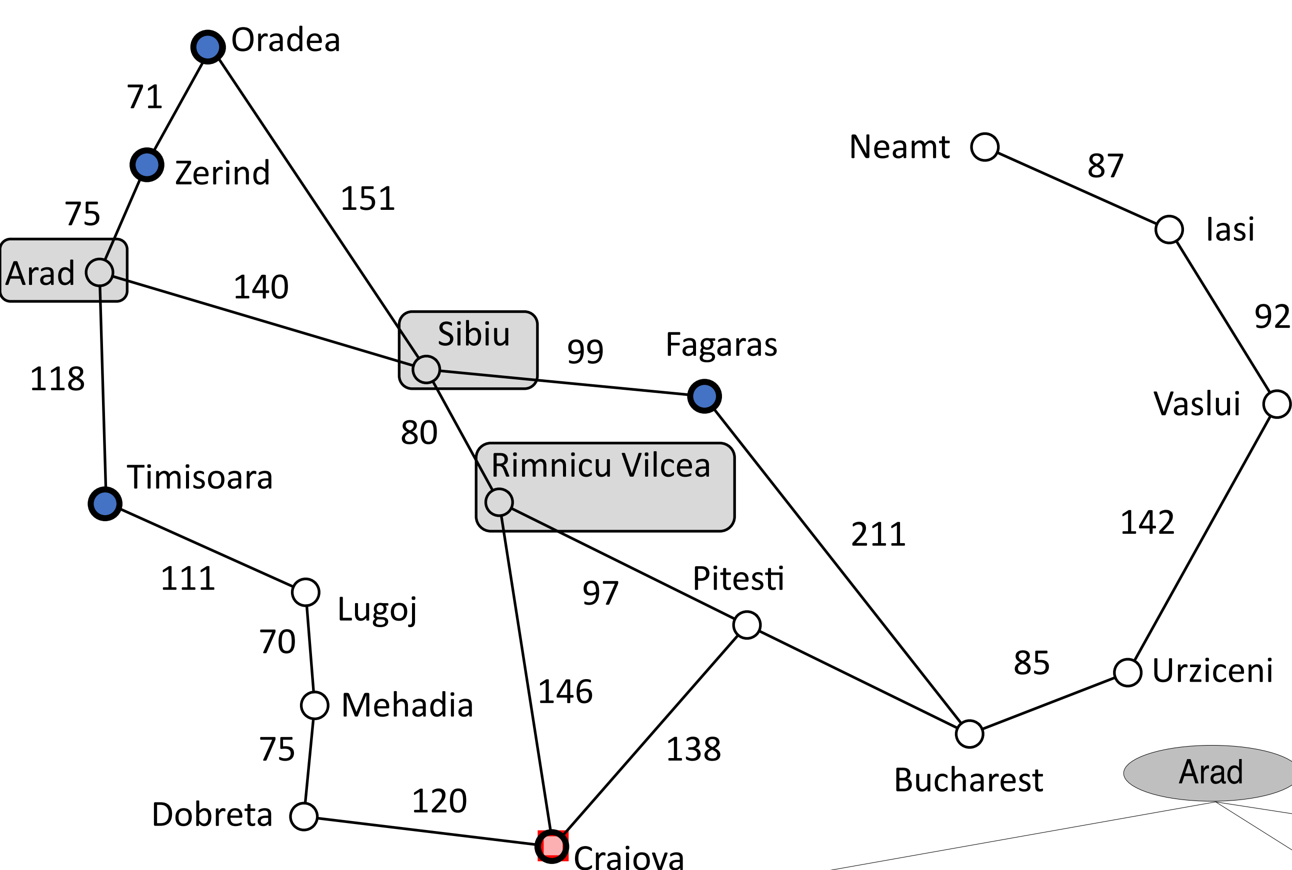


straight-line dist. from s to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Fagaras	176
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

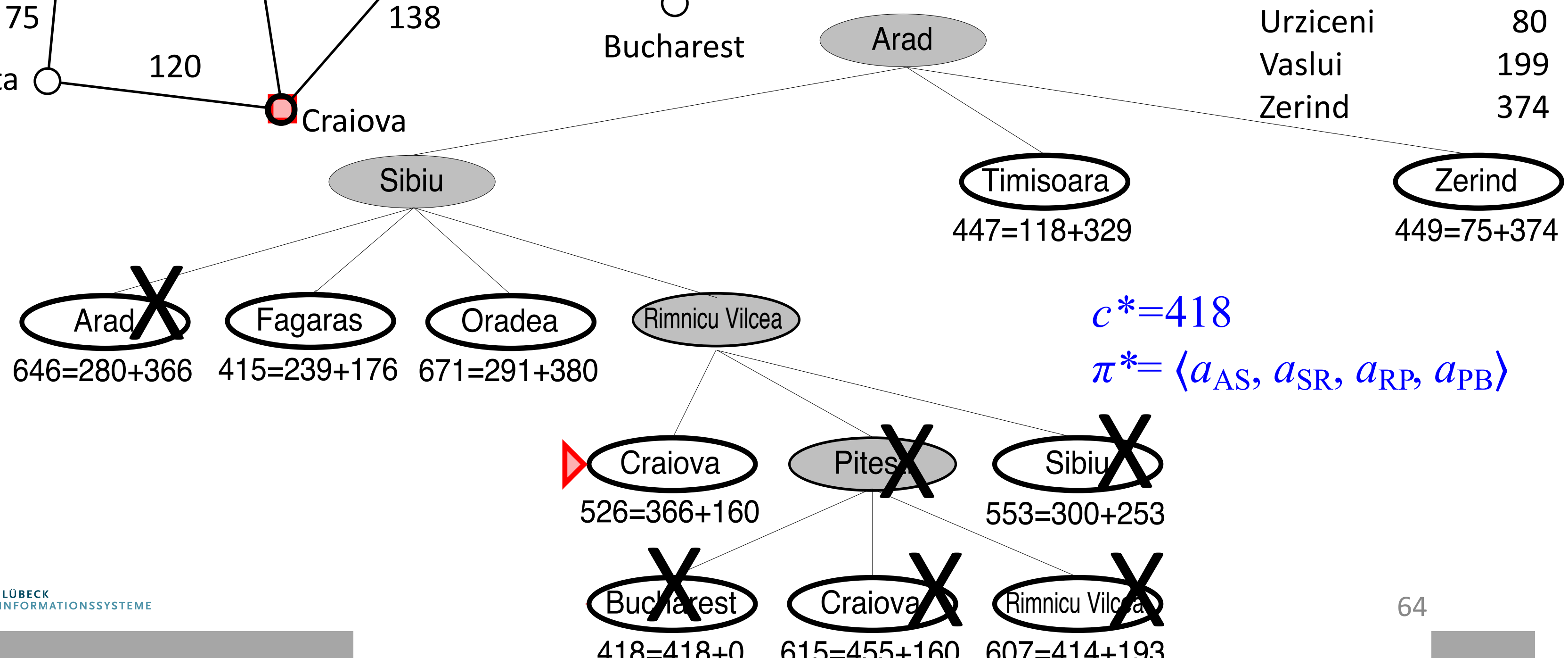


$c^*=418$

$\pi^* = \langle a_{AS}, a_{SR}, a_{RP}, a_{PB} \rangle$

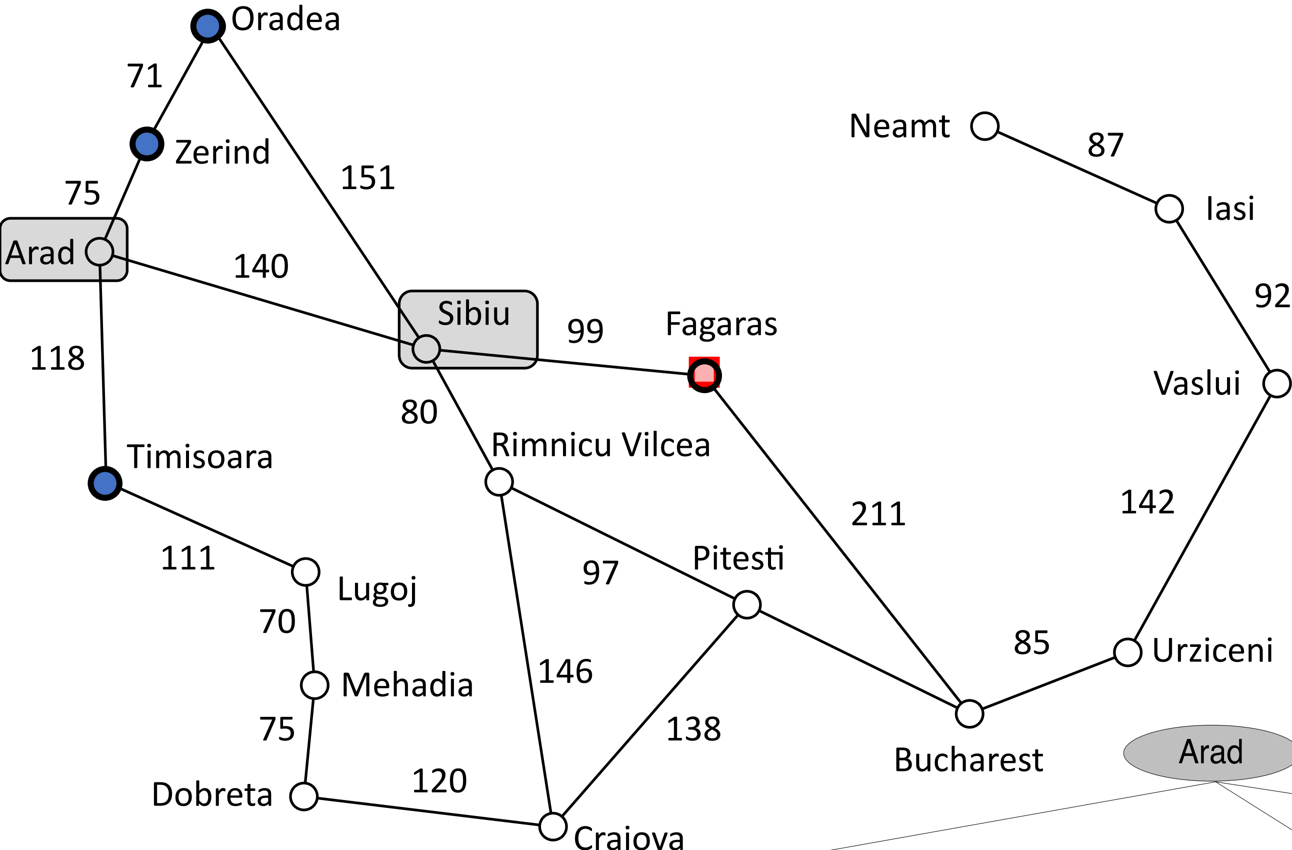


straight-line dist. from s to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Fagaras	176
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

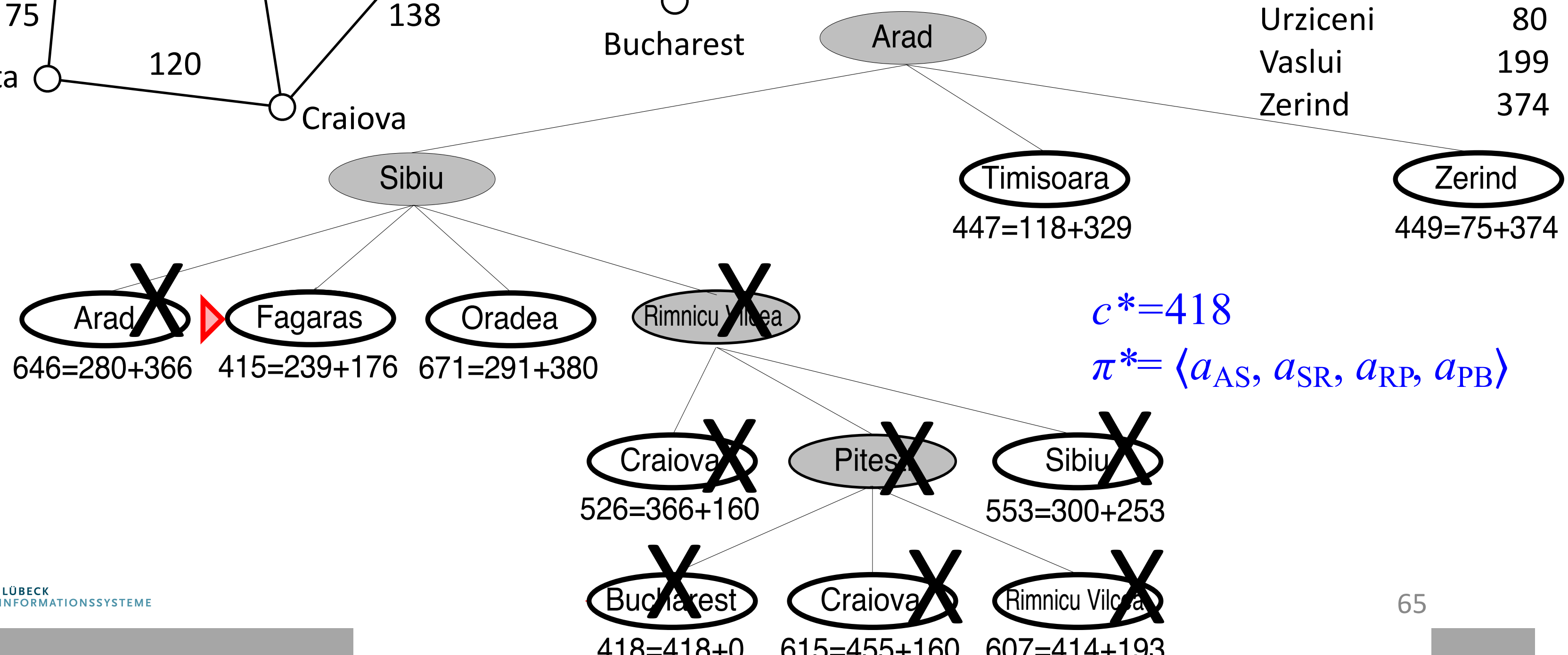


$c^*=418$

$\pi^*= \langle a_{AS}, a_{SR}, a_{RP}, a_{PB} \rangle$

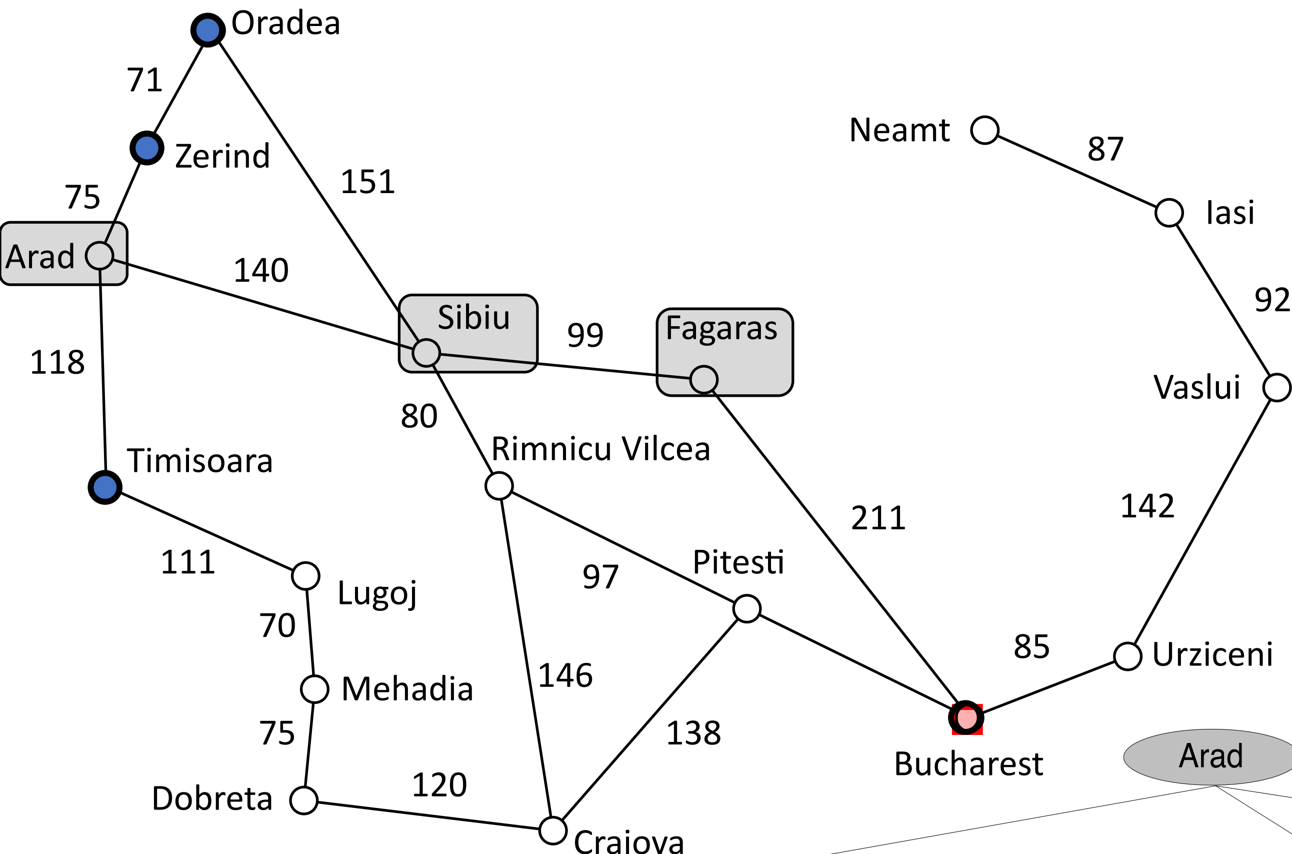


straight-line dist. from s to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Fagaras	176
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

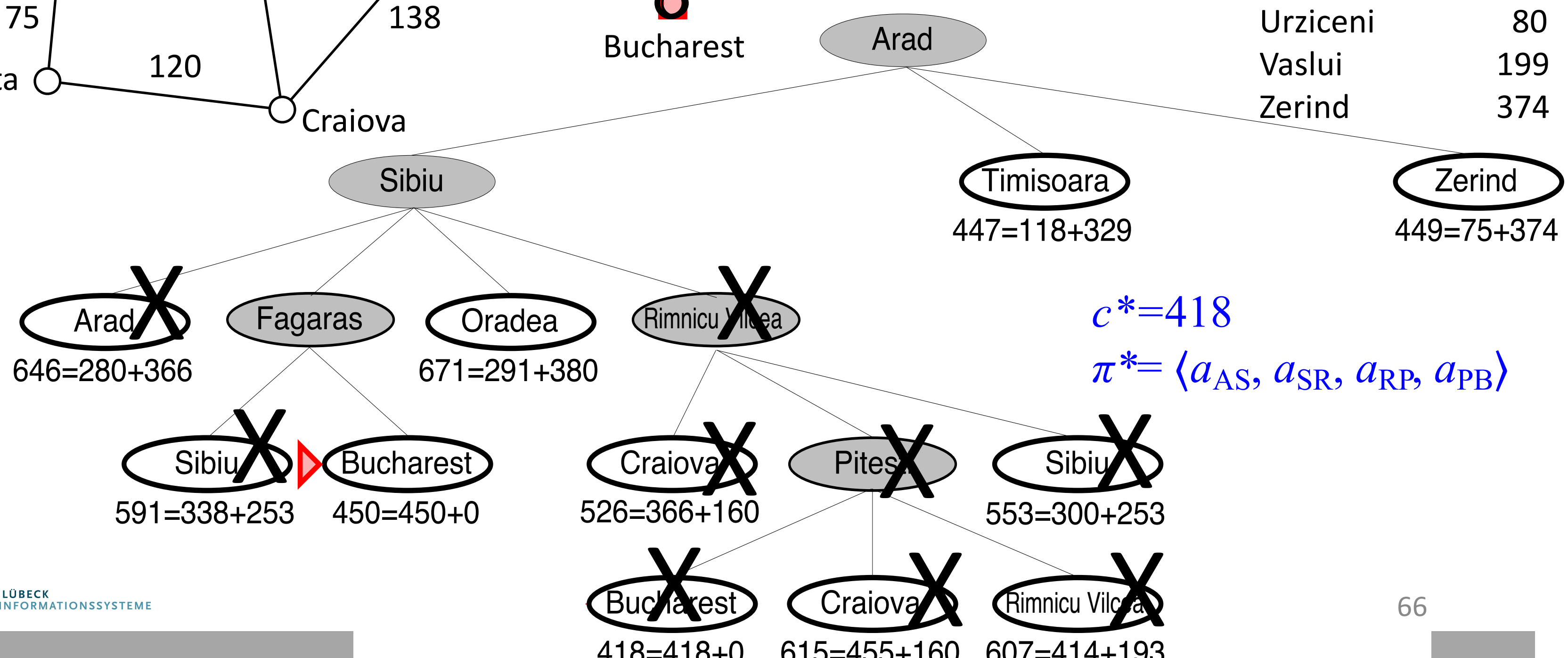


$c^{*}=418$

$\pi^{*}=\langle a_{\mathrm{AS}}, a_{\mathrm{SR}}, a_{\mathrm{RP}}, a_{\mathrm{PB}}\rangle$



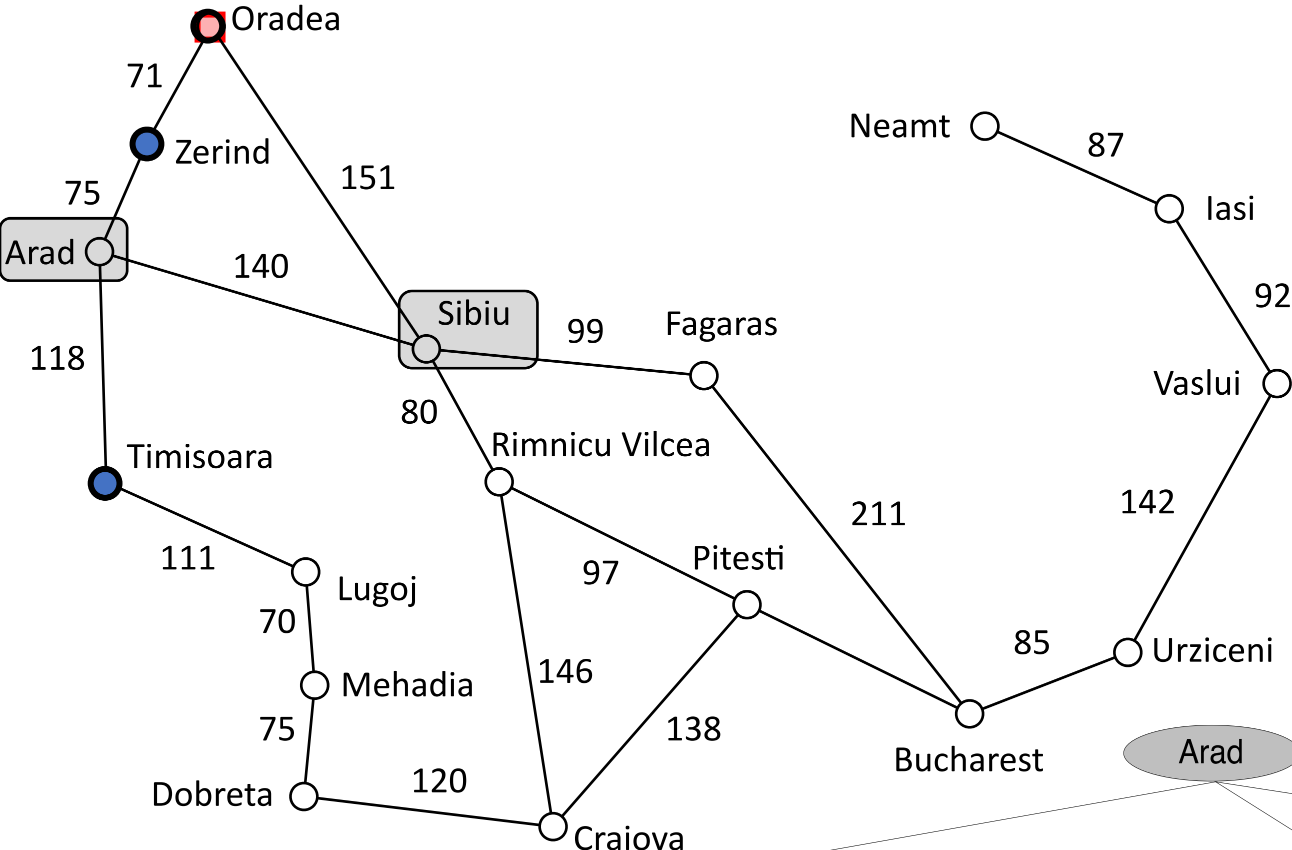
straight-line dist. from s to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Fagaras	176
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

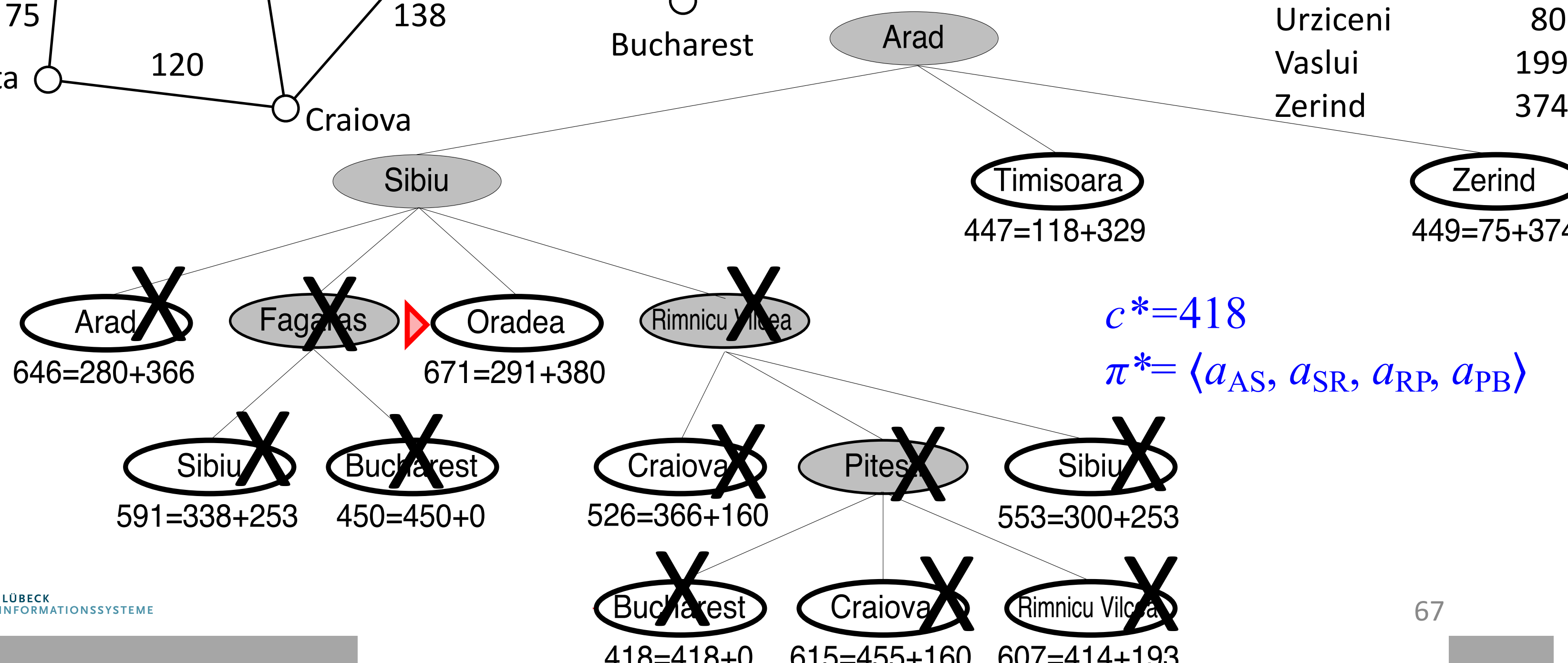


$c^*=418$

$\pi^* = \langle a_{\mathrm{AS}}, a_{\mathrm{SR}}, a_{\mathrm{RP}}, a_{\mathrm{PB}} \rangle$

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





$c^* = 418$

$\pi^* = \langle a_{AS}, a_{SR}, a_{RP}, a_{PB} \rangle$

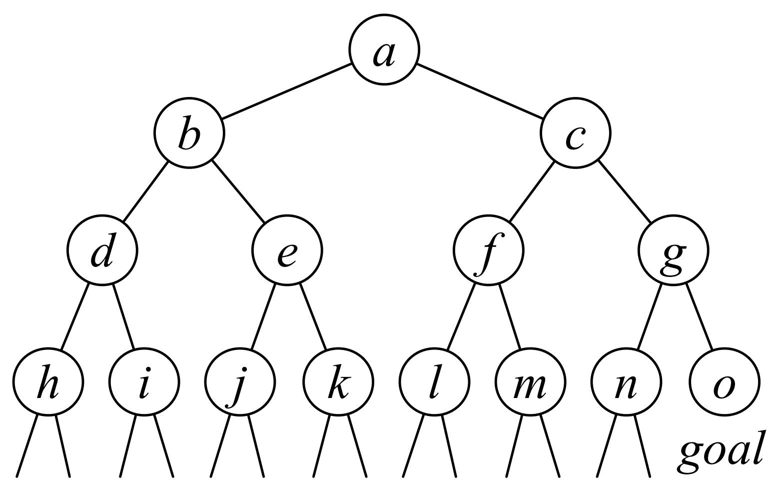
Iterative Deepening Search (IDS)

- Example:
  - Expand a
    - $(k = 1)$
  - Expand a, b, c
    - $(k = 2)$
  - Expand a, b, c, d, e, f, g
    - $(k = 3)$
  - Expand $a, b, c, d, e, f, g, h, i, j, k, l, m, n, o$
    - $(k = 4)$
  - Solution path $\langle a, c, g, o \rangle$
  - Total number of node expansions:
    - $1 + 3 + 7 + 15 = 26$
- If goal is at depth d and branching factor is 2:

$$\sum_{i=1}^{d} (2^i - 1) = \left(\sum_{i=1}^{d} 2^i\right) - d$$

$$= 2^{d+1} - 2 - d = O(2^d)$$

```
IDS(Σ, s0, g)
for k = 1 to ∞ do
    π* ← do DFS, backtracking at every
            node of depth k
    if π* ≠ failure then
        return π*
    if the search generated
            no nodes of depth k then
        return failure
```



Iterative Deepening Search (IDS)

- If goal is at depth d and branching factor is b:

$$\sum_{i=1}^{d}\left(b^i - 1\right) = \left(\sum_{i=1}^{d} b^i\right) - d$$

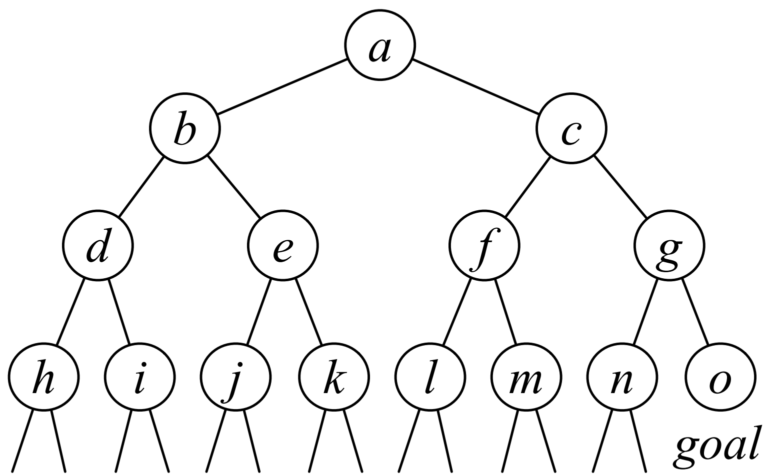
$$= b^{d+1} - b - d = O\left(b^d\right)$$

- Properties
  - Termination, completeness, optimality
    - same as BFS
  - Worst-case complexity
    - Memory requirement $O(bd)$
      - vs. $O(b^d)$ with BFS
    - Worst-case running time $O(b^d)$
      - vs. $O(b^l)$ for DFS
      - If the number of nodes at depth d grows exponentially with d

where

- b = max branching factor
- d = min solution depth if there is one, otherwise max depth of any node

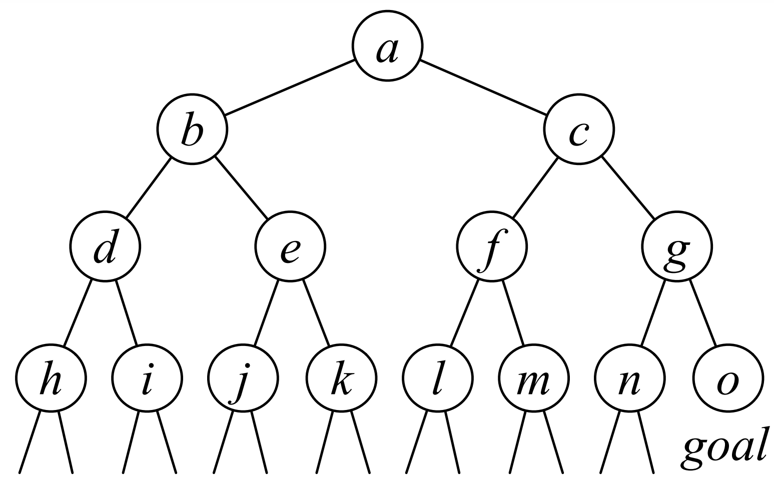
```
IDS(Σ, s₀, g)
for k = 1 to ∞ do
    π* ← do DFS, backtracking at every
            node of depth k
    if π* ≠ failure then
        return π*
    if the search generated
            no nodes of depth k then
        return failure
```



IDA*

- Properties
  - Termination, completeness, and optimality same as A*
  - Worst-case complexity
    - If h is admissible, memory requirement $O(bd)$ rather than $O(b^d)$
    - If the number of nodes grows exponentially with c, running time $O(b^d)$
      - Can be much worse if the number of nodes grows subexponentially
        - e.g., real-valued costs
- IDA* is not much used in practice

```
IDA*(Σ, s_0, g)
c ← 0
loop
    π* ← do DFS, backtracking whenever
            f(ν) > c
    if π* ≠ failure then
        return π*
    if DFS didn't generate
            an f(ν) > c then
        return failure
    c ← the smallest f(ν) > c
            where backtracking occurred
```



Discussion

- If h is admissible, both A* and DFBB will return optimal solutions
  - Usually DFBB takes more time, A* takes more memory
  - A* better than DFBB in highly connected graphs (many paths to states)
    - DFBB can have exponentially worse running time than A*
  - 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* takes exponentially more memory than DFBB
- DFS returns the first solution it finds
  - Less backtracking than DFBB, but solution can be very far from optimal
- GBFS returns the first solution it finds
  - With a good heuristic function, usually near-optimal without much backtracking
  - Used by most classical planners nowadays

Intermediate Summary

- Forward-search, Deterministic-Search
- Cycle-checking
- Breadth-first, depth-first, uniform-cost search
- A*, GBFS, DFBB
- IDS, IDA*

Outline per the Book

2.1 *State-variable representation*

- State = {values of variables}; action = changes to those values

2.2 *Forward state-space search*

- Start at initial state, look for sequence of actions that achieve goal

2.3 *Heuristic functions*

- How to guide a forward state-space search

2.6 *Incorporating planning into an actor*

- Online lookahead, unexpected events

2.4 *Backward search*

- Start at goal state, go backwards toward initial state

2.5 *Plan-space search*

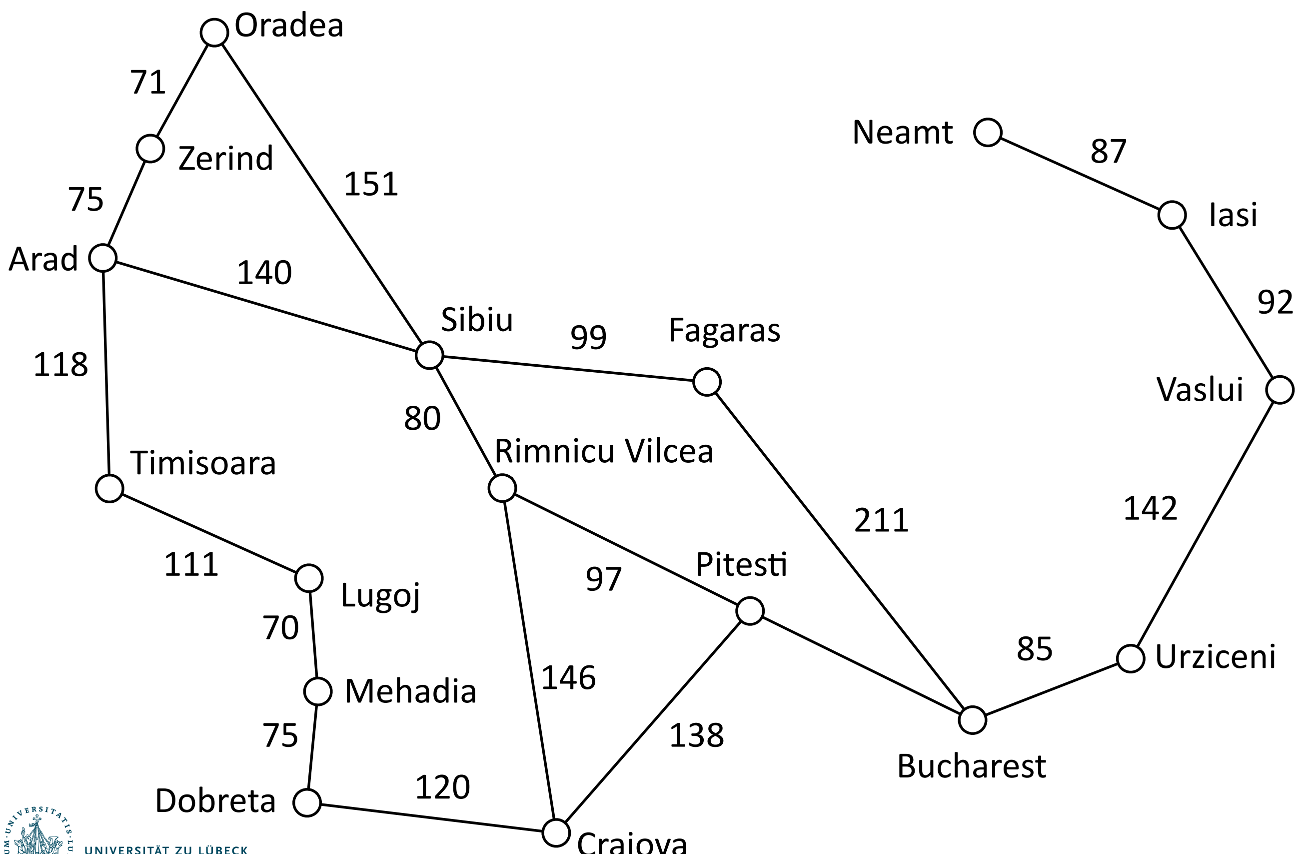
- Start with incomplete plan for getting from initial state to goal state, make transformations to fix flaws in the plan

Heuristic Functions

- Planning problem P in domain Σ
- Creating a heuristic function:
  - Weaken some of the constraints that
    - restrict what the states, actions, and plans are
    - restrict when an action or plan is applicable, what goals it achieves
    - increase the costs of actions and plans
- Relaxed planning domain $\Sigma' = (S', A', \gamma')$ and problem $P' = (\Sigma', s_0', g')$
  - for every solution π for P, P' has a solution π' with $cost'(\pi') \leq cost(\pi)$
- Suppose we have an algorithm A for solving planning problems in Σ'
  - Heuristic function $h^A(s)$ for P:
    - Find a solution π' for (Σ', s, g'); return $cost(\pi')$
  - If A runs quickly, then h^A may be a useful heuristic function
  - If A always finds optimal solutions, then h^A is admissible

Example from A*

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



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

Domain-independent Heuristics

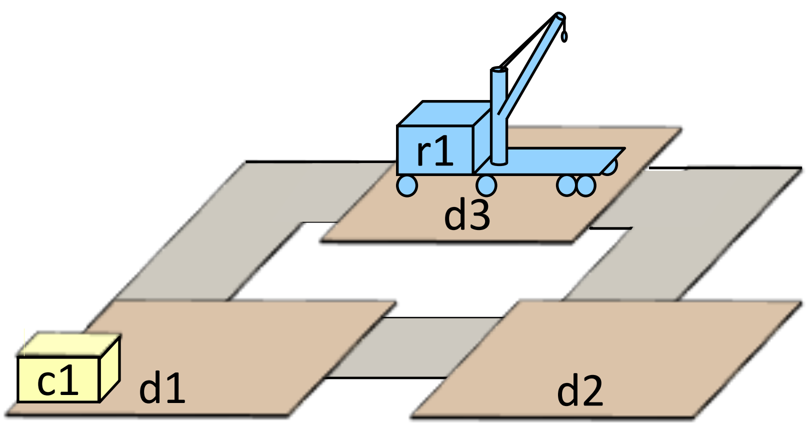
- Heuristic functions that can be used work in any classical planning problem
  - Additive-cost heuristic
  - Max-cost heuristic
  - Delete-relaxation heuristics
    - Optimal relaxed solution
    - Fast-forward heuristic
  - Landmark heuristics

In the book, but I'll skip them

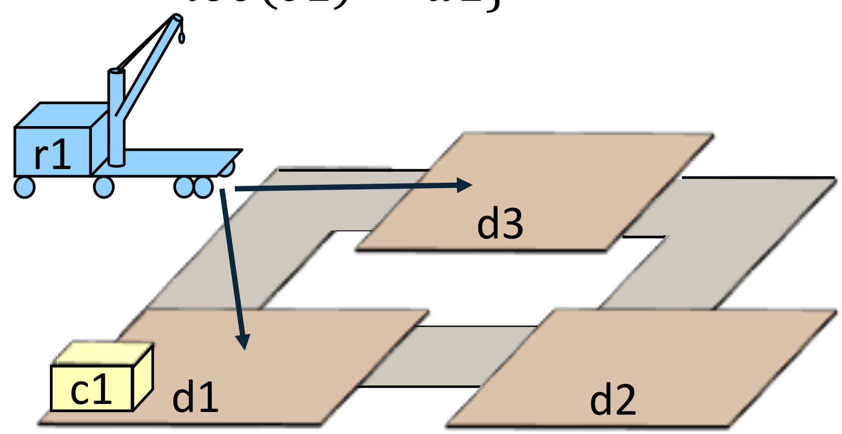
Delete-Relaxation

- Relaxation:
  - A state variable can have more than one value at the same time
  - When assigning a new value, keep the old one too
- Suppose state s includes an atom $x = v$, action a has effect $x \leftarrow w$
  - $\gamma^+(s, a)$ is a relaxed state
  - Includes both $x = v$ and $x = w$

- Example
  - $s_0 = \{loc(r1) = d3, cargo(r1) = nil, loc(c1) = d1\}$
  - $move(r1, d3, d1)$
    - Pre: $loc(r1) = d3$
    - Eff: $loc(r1) \leftarrow d1$
  - $\hat{s}_1 = \gamma^+(s_0, move(r1, d3, d1))$
    $= \{loc(r1) = d3,$
    $loc(r1) = d1,$
    $cargo(r1) = nil,$
    $loc(c1) = d1\}$



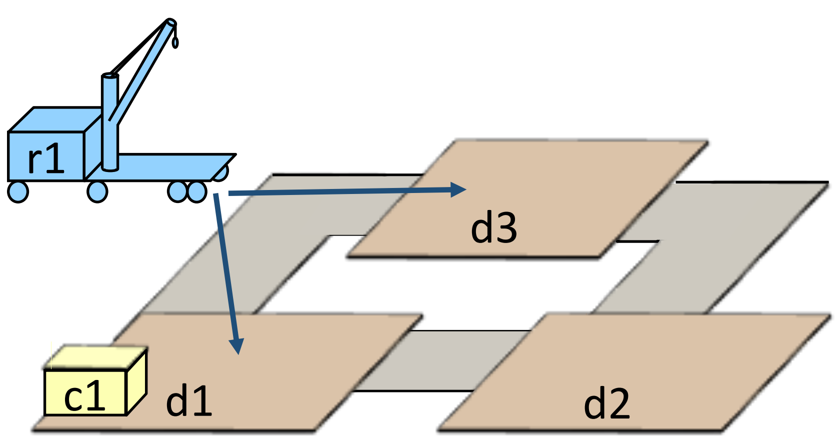
s_0



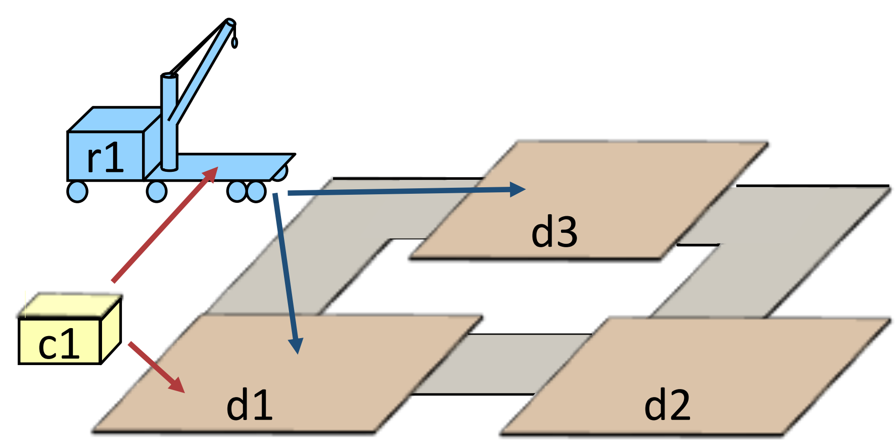
$\hat{s}$

Relaxed States

- Relaxed state (or *r-state*):
  - Set $\hat{s}$ of ground atoms that includes at least 1 value for each state variable
  - Represents {all states that are subsets of $\hat{s}$}
  - Note: every state s is also a relaxed state that represents $\{s\}$
- Examples
  - $\hat{s}_1 = \{loc(r1) = d1, loc(r1) = d3, cargo(r1) = nil, loc(c1) = d1\}$
  - $\hat{s}_2 = \gamma^+\big(\hat{s}_1, take(r1, d1, c1)\big) = \{loc(r1) = d1, loc(r1) = d3,$ $cargo(r1) = nil\,, loc(c1) = r1, loc(c1) = d1, cargo(r1) = c1\}$



$\hat{s}_1$



$\hat{s}_2$

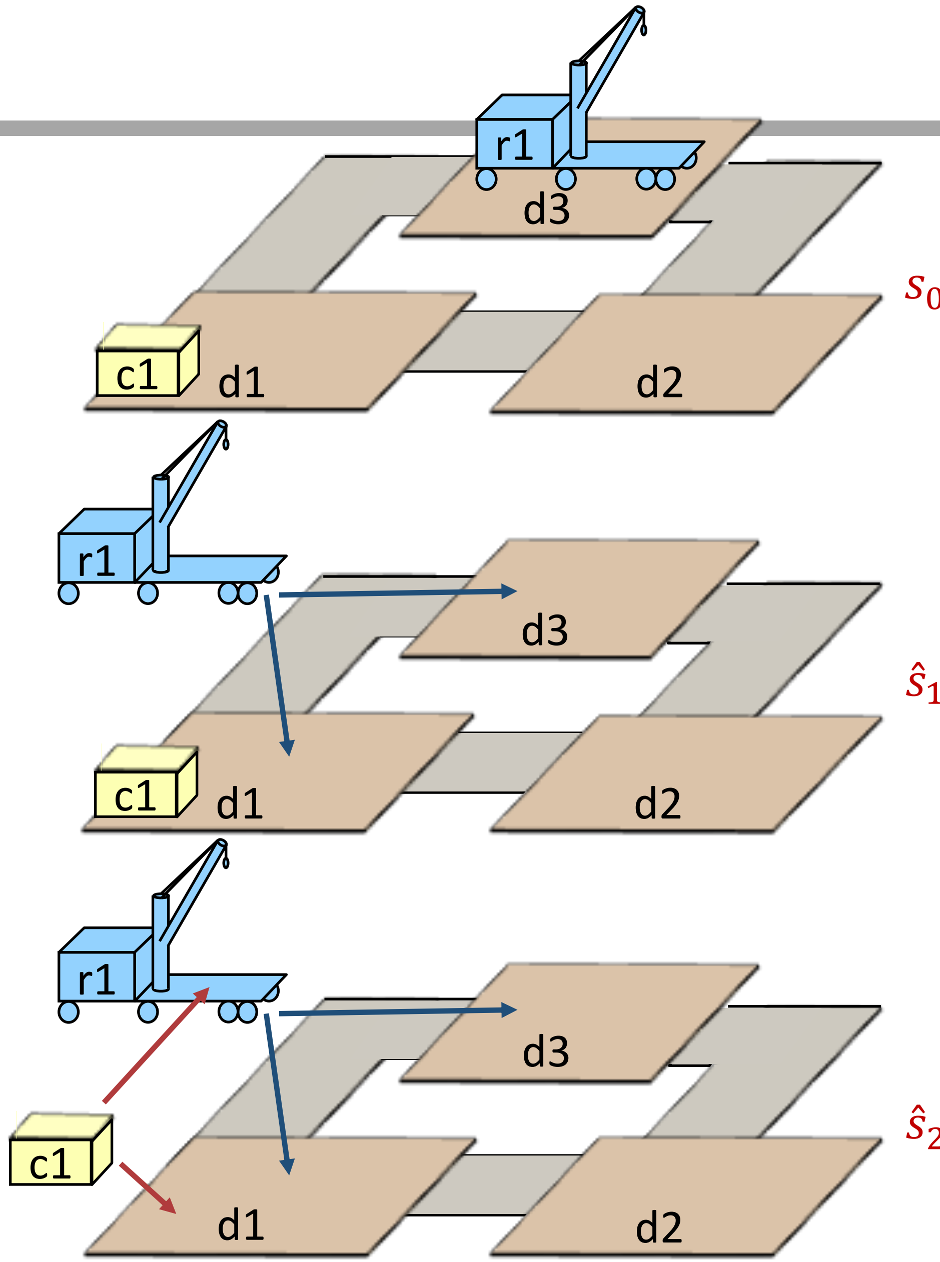
R-applicability

- An r-state $\hat{s}$ r-satisfies a set of literals g if a set $s \subseteq \hat{s}$ satisfies g
- Action a is r-applicable in $\hat{s}$ if $\hat{s}$ r-satisfies $pre(a)$
  - i.e., $\hat{s}$ contains a subset s that satisfies the preconditions of a
  - If a is r-applicable, then $\gamma^+(\hat{s}, a) = \hat{s} \cup \gamma(s, a)$

- $\pi = \langle a_1, \ldots, a_n \rangle$ is r-applicable in $\hat{s}_0$ if there are r-states $\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_n$ such that
  - a_1 is r-applicable in $\hat{s}_0$ and $\gamma^+(\hat{s}_0, a_1) = \hat{s}_1$
  - a_2 is r-applicable in $\hat{s}_1$ and $\gamma^+(\hat{s}_1, a_2) = \hat{s}_2$
  - …
  - a_n is r-applicable in $\hat{s}_{n-1}$ and $\gamma^+(\hat{s}_{n-1}, a_n) = \hat{s}_n$
- In this case, $\gamma^+(\hat{s}_{n-1}, \pi) = \hat{s}_n$

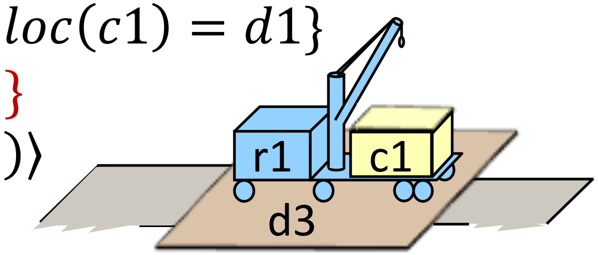
Example

- $s_0 = \{loc(r1) = d3,$
  $cargo(r1) = nil, loc(c1) = d1\}$
- $move(r1, d3, d1)$
  - Pre: $loc(r1) = d3$
  - Eff: $loc(r1) \leftarrow d1$
- $\hat{s}_1 = \gamma^+(\hat{s}_1, move(r1, d1, c1)) =$
  $\{loc(r1) = d1, loc(r1) = d3,$
  $cargo(r1) = nil, loc(c1) = d1\}$
- $take(r, l, c)$
  - pre: $cargo(r) = nil, loc(r) = l,$
    $loc(c) = l$
  - eff: $cargo(r) \leftarrow c, loc(c) \leftarrow r$
- $\hat{s}_2 = \gamma^+(\hat{s}_1, take(r1, d1, c1)) =$
  $\{loc(r1) = d1, loc(r1) = d3,$
  $cargo(r1) = nil\ , loc(c1) = r1,$
  $loc(c1) = d1, cargo(r1) = c1\}$



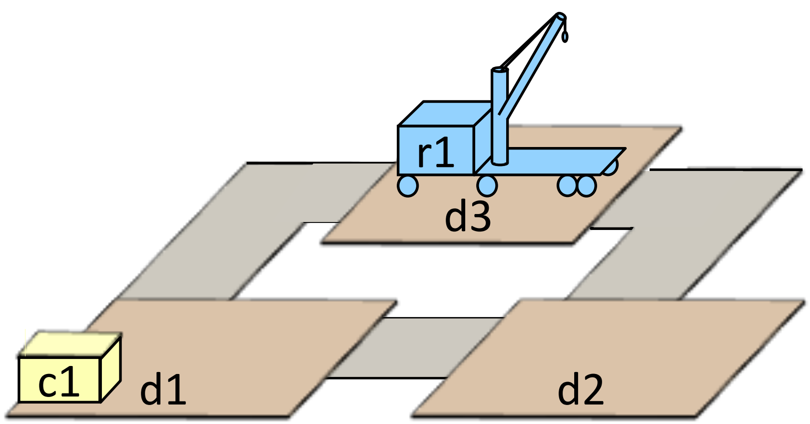
Relaxed Solution

- Planning problem $P = (\Sigma, s_0, g)$
- Plan π is a relaxed solution for P if $\gamma^+(\hat{s}_0, \pi)$ r-satisfies g
- Example:
  - Initial $s_0 = \{loc(r1) = d3, cargo(r1) = nil, loc(c1) = d1\}$
  - Goal states $g = \{loc(r1) = d3, loc(c1) = r1\}$
  - Plan $\pi \ = \langle move(r1, d3, d1), take(r1, c1, d1)\rangle$

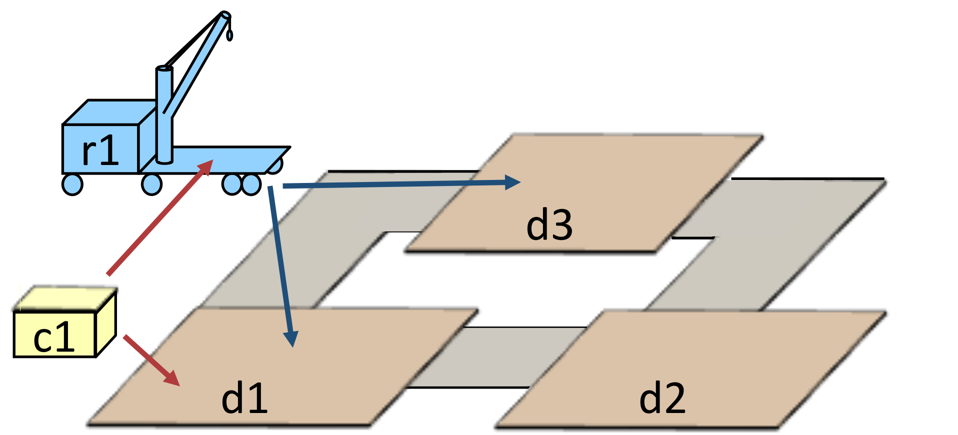


g

  - End state $\gamma^+(s_0, \pi) \ = \ \{loc(r1) = d1, loc(r1) = d3,$ $cargo(r1) = nil \ , loc(c1) = r1, loc(c1) = d1, cargo(r1) = c1\}$



s_0



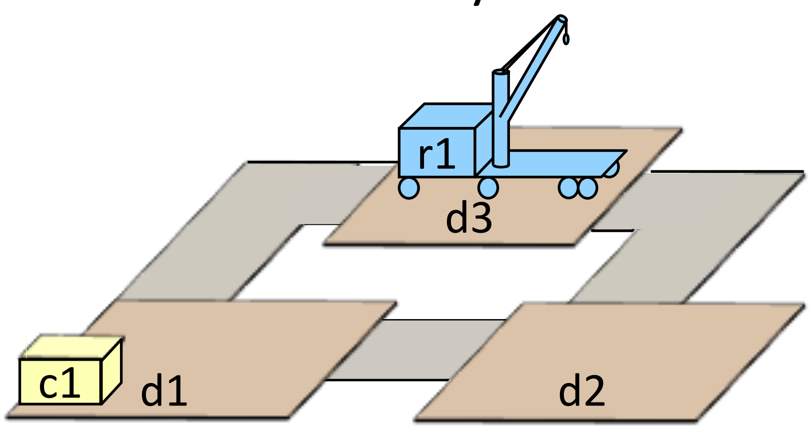
$\hat{s}_2$

Optimal Relaxed Solution Heuristic

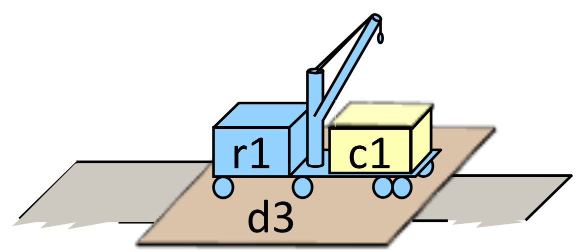
- Given a planning problem $P = (\Sigma, s_0, g)$
- **Optimal relaxed solution heuristic**:
  - $h^+(s)$ = minimum cost of all relaxed solutions for P

How does this compare with $h^*(s)$?

- Example:
  - Initial $s_0 = \{loc(r1) = d3,\ cargo(r1) = nil, loc(c1) = d1\}$
  - Goal states $g = \{loc(r1) = d3, loc(c1) = r1\}$
  - $\pi\ = \langle move(r1, d3, d1), take(r1, c1, d1)\rangle$
    - $cost(\pi) = 2$
  - No less-costly relaxed solution, so $h^+(s_0) = 2$



s_0

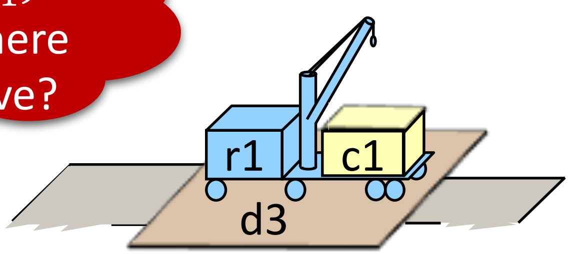


g

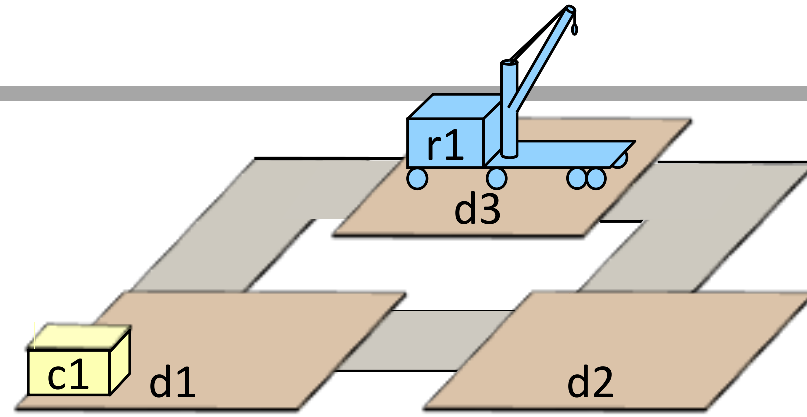
Example

- $s_0 = \{loc(r1) = d3,$ $cargo(r1) = nil, loc(c1) = d1\}$
- In s_0, two applicable actions
  - $a_1 = move(r1, d3, d1)$
    - $s_1 = \{loc(r1) = d1, cargo(r1) = nil,\ loc(c1) = d1\}$
  - $a_2 = move(r1, d3, d2)$
    - $s_2 = \{loc(r1) = d2, cargo(r1) = nil,\ loc(c1) = d1\}$
- GBFS evaluates $h^+(s_1)$ and $h^+(s_2)$, and chooses to move to whichever is smaller

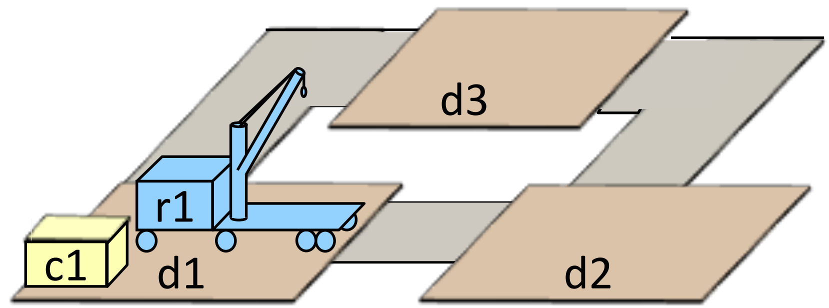
What are $h^+(s_1)$ and $h^+(s_2)$ and where does GBFS move?



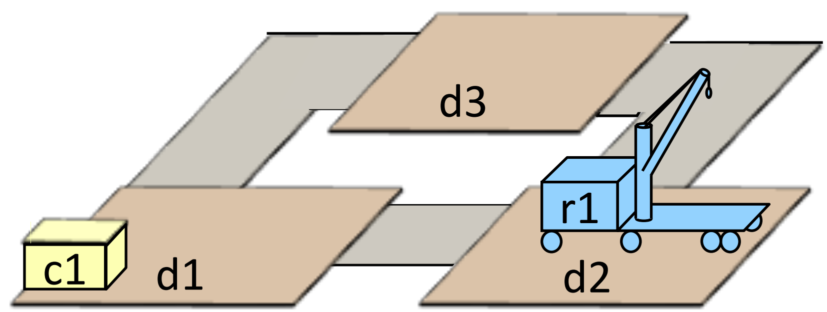
$g = \{loc(r1) = d3, loc(c1) = r1\}$



s_0



s_1



s_2

Fast-Forward Heuristic

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

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

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

Preliminaries

- Let A_1 be a set of actions that are r-applicable in $\hat{s}$
  - Can apply them in any order and get same result
  - Define result of applying A_1 in $\hat{s}$ as

$$\gamma^+(\hat{s}, A_1) = \hat{s} \cup \bigcup_{a \in A_1} eff(a)$$

- Let $\hat{s}_1 = \gamma^+(\hat{s}_0, A_1)$
  - Suppose A_2 is a set of actions that are r-applicable in $\hat{s}_1$
  - Define $\gamma^+(\hat{s}_0, \langle A_1, A_2 \rangle) = \gamma^+(\hat{s}_1, A_2)$
  - ...
  - Define $\gamma^+(\hat{s}_0, \langle A_1, A_2, \dots, A_n \rangle)$ in the obvious way

Fast-Forward Heuristic

```
HFF(Σ,s,g)
    // construct a relaxed solution ⟨A1,A2,…,Ak⟩:
    ŝ0 ← s
    for k = 1; k++; a subset of ŝk r-satisfies g do
        Ak = {all actions r-applicable in ŝk-1}
        ŝk = γ+(sk-1, Ak)
        if k > 1 and ŝk = ŝk-1 then
            return ∞                                  // there's no solution
    // extract minimal relaxed solution ⟨â1, â2, …, âk⟩:
    ĝk = g
    for i = k down to 1 do
        âi = any minimal subset of Ai such that γ+(ŝi-1,âi) r-satisfies ĝi
        ĝi-1 ← (ĝi \ eff(âi)) ∪ pre(âi)
    π̂ ← ⟨â1,…,âk⟩
    return ∑a is an action in π̂ cost(a)              // upper bound on h+
```

- Find a minimal relaxed solution and return its cost
  - Generates a sequence of successively larger r-states and sets of applicable actions until $\hat{s}_k$ r-satisfies g:
    $$\hat{s}_0, A_1, \hat{s}_1, A_2, \hat{s}_2, \dots, A_{k-1}, \hat{s}_{k-1}, A_k, \hat{s}_k$$
  - Extract minimal relaxed solution from that sequence

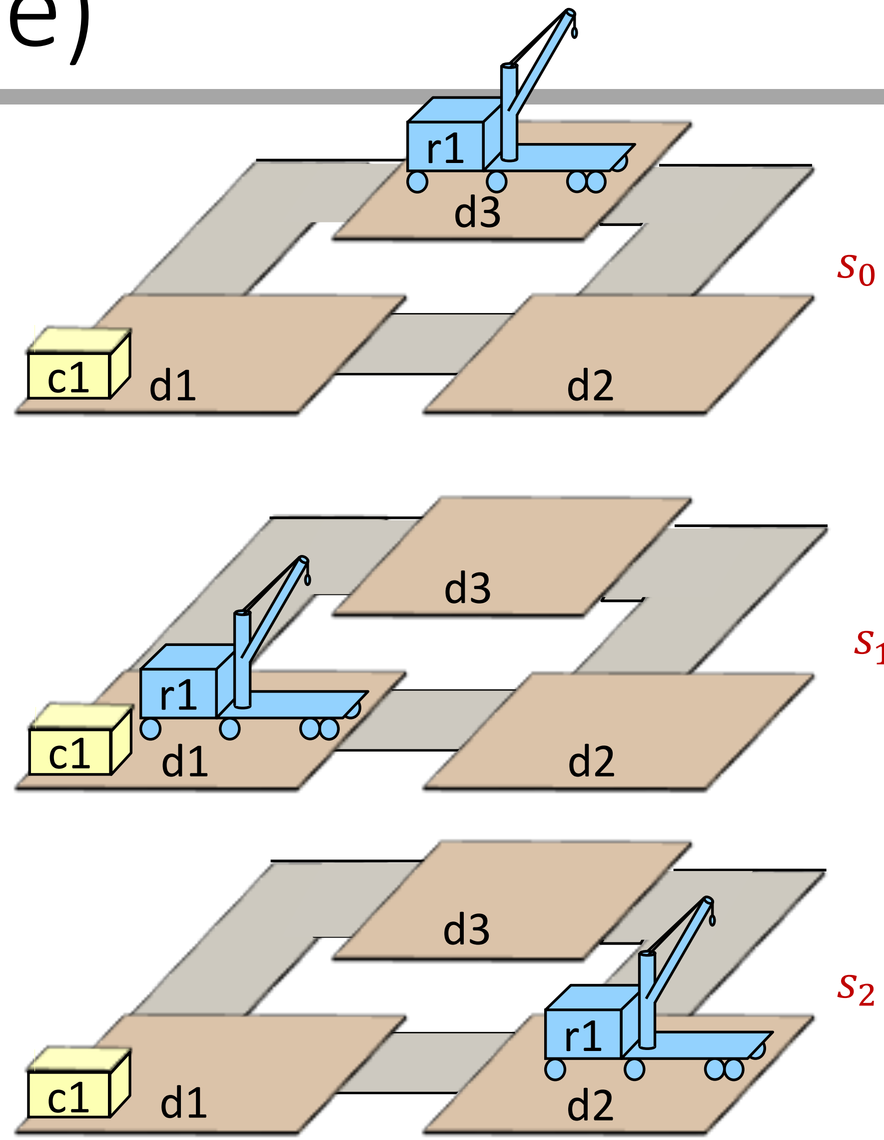
Fast-Forward Heuristic

```
HFF(Σ,s,g)
    // construct a relaxed solution ⟨A₁,A₂,…,Aₖ⟩:
    ŝ₀ ← s
    for k = 1; k++; a subset of ŝₖ r-satisfies g do
        Aₖ = {all actions r-applicable in ŝₖ₋₁}
        ŝₖ = γ⁺(sₖ₋₁, Aₖ)
        if k > 1 and ŝₖ = ŝₖ₋₁ then
            return ∞                              // there's no solution
    // extract minimal relaxed solution ⟨â₁, â₂, …, âₖ⟩:
    ĝₖ = g
    for i = k down to 1 do
        âᵢ = any minimal subset of Aᵢ such that γ⁺(ŝᵢ₋₁,âᵢ) r-satisfies ĝᵢ
        ĝᵢ₋₁ ← (ĝᵢ \ eff(âᵢ)) ∪ pre(âᵢ)
    π̂ ← ⟨â₁,…,âₖ⟩
    return ∑_{a is an action in π̂} cost(a)        // upper bound on h⁺
```

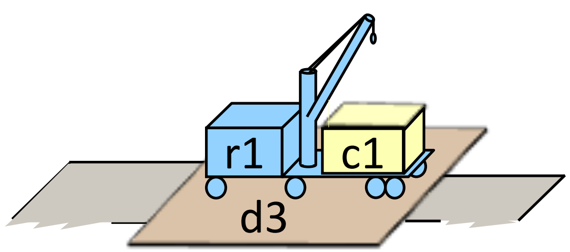
- Find a minimal relaxed solution and return its cost
- Define h^{FF} = the value returned by HFF(Σ, s, g)
  - Return value is ambiguous
    - Each $\hat{a}_i$ in $h^{FF}(s)$ is a minimal set of actions s.t. $\gamma^+(\hat{s}_{i-1}, \hat{a}_i)$ r-satisfies $pre(\hat{a}_i)$
    - Depends on *which* minimal subsets we choose

Example (as before)

- $s_0 = \{loc(r1) = d3,$ $cargo(r1) = nil, loc(c1) = d1\}$
- In s_0, two applicable actions
  - $a_1 = move(r1, d3, d1)$
    - $s_1 = \{loc(r1) = d1, cargo(r1) = nil,\ loc(c1) = d1\}$
  - $a_2 = move(r1, d3, d2)$
    - $s_2 = \{loc(r1) = d2, cargo(r1) = nil,\ loc(c1) = d1\}$



- GBFS using h^{FF}
  - Compute $h^{FF}(s_1)$ and $h^{FF}(s_2)$
  - Move to whichever is smaller



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

Example

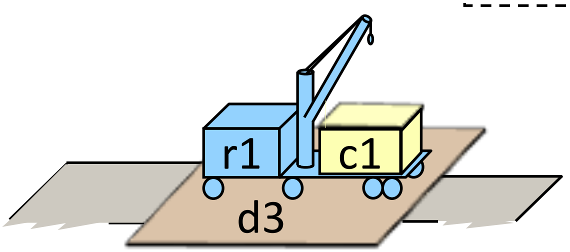
```
// construct a relaxed solution ⟨A_1, A_2, …, A_k⟩:
ŝ_0 ← s
for k = 1; k++; subset of ŝ_k r-satisfies g do
    A_k = {all actions r-applicable in ŝ_{k-1}}
    ŝ_k = γ^+(s_{k-1}, A_k)
    if k > 1 and ŝ_k = ŝ_{k-1} then
        return ∞            // there's no solution
```

Relaxed Planning Graph (RPG) from $\hat{s}_0 = s_1$ to g
(solid lines indicate preconditions/effects):

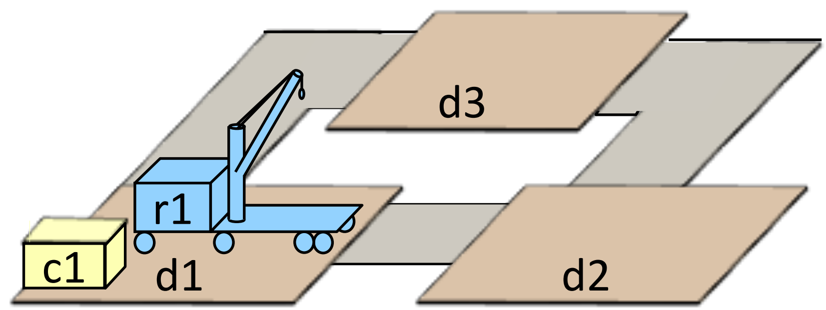
Atoms in $\hat{s}_0 = s_1$:	Actions in A_1:	Atoms in $\hat{s}_1$:
$loc(r1) = d1$	$move(r1, d1, d2)$	$loc(r1) = d2$
$loc(c1) = d1$	$move(r1, d1, d3)$	$loc(r1) = d3$
$cargo(r1) = nil$	$take(r1, c1, d1)$	$loc(c1) = r1$
		$cargo(r1) = c1$
	from $\hat{s}_0$:	$loc(r1) = d1$
		$loc(c1) = d1$
		$cargo(r1) =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

$\langle A_1 \rangle$ is a relaxed solution

$\gamma^+(s_0, A_1)$ r-satisfies g



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



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

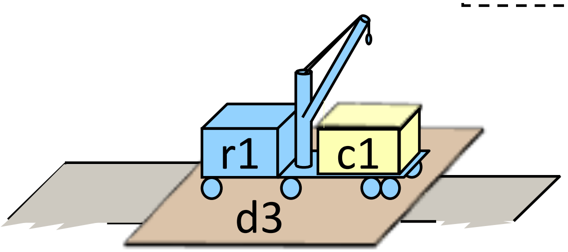
Example

```
// extract minimal relaxed solution
ĝ_k = g
for i = k down to 1 do
    â_i = minimal subset of A_i s.t. γ⁺(ŝ_{i-1},â_i) r-satisfies ĝ_i
    ĝ_{i-1} ← (ĝ_i \ eff(â_i)) ∪ pre(â_i)
```

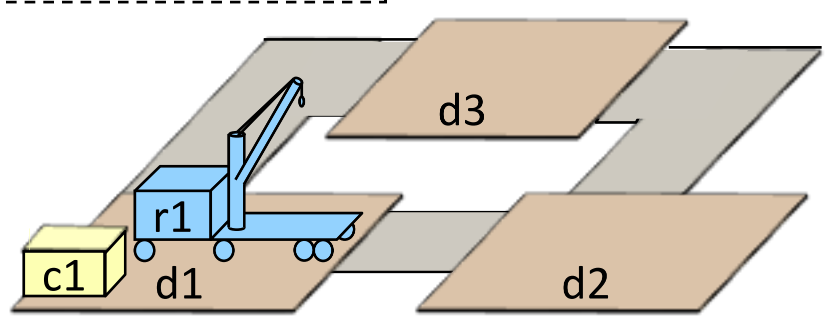
- $\langle \hat{a}_1 \rangle$ is a minimal relaxed solution
- Cost of each action is 1, so $h^{FF}(s_1) = 2$

Relaxed Planning Graph (RPG) from $\hat{s}_0 = s_1$ to g
(follow lines from atoms in g: all if precond.; one if eff.)

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



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



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

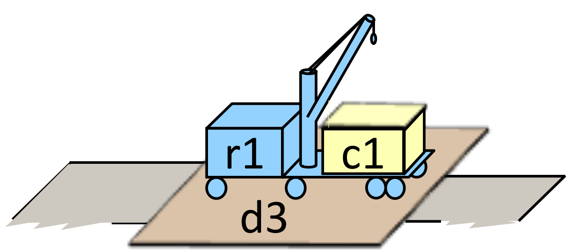
Example

```
// construct a relaxed solution ⟨A₁, A₂, …, Aₖ⟩:
ŝ₀ ← s
for k = 1; k++; subset of ŝₖ r-satisfies g do
    Aₖ = {all actions r-applicable in ŝₖ₋₁}
    ŝₖ = γ⁺(sₖ₋₁, Aₖ)
    if k > 1 and ŝₖ = ŝₖ₋₁ then
        return ∞              // there's no solution
```

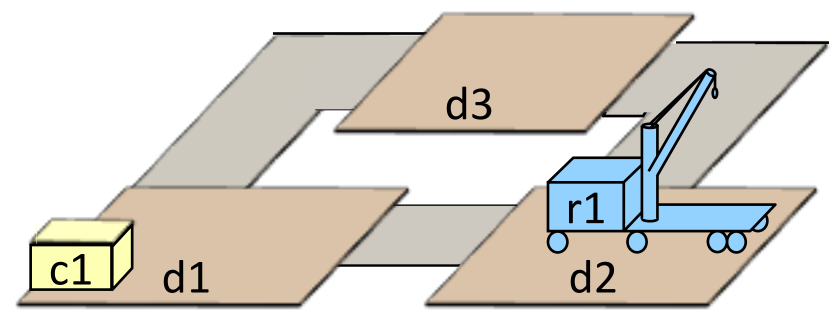
RPG from $\hat{s}_0 = s_2$ to g

Atoms in $\hat{s}_0 = s_2$:	Actions in A_1:	Atoms in $\hat{s}_1$:	Actions in A_2:	Atoms in $\hat{s}_2$:
$loc(r1) = d2$	$move(r1, d2, d3)$	$loc(r1) = d3$	$move(r1, d3, d2)$	$loc(r1) = d2$
$loc(c1) = d1$	$move(r1, d2, d1)$	$loc(r1) = d1$	$move(r1, d1, d2)$	$loc(c1) = d1$
$cargo(r1) = nil$		from $\hat{s}_0$: $loc(r1) = d2$	$move(r1, d3, d1)$	$cargo(r1) = nil$ (from $\hat{s}_1$)
		$loc(c1) = d1$	$move(r1, d1, d3)$	$loc(r1) = d1$
		$cargo(r1) = nil$	$move(r1, d2, d1)$	$loc(r1) = d3$
			$move(r1, d2, d3)$	$cargo(r1) = c1$
			$take(r1, c1, d1)$	$loc(c1) = r1$

$\langle A_1, A_2 \rangle$ is a relaxed solution



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



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

Example

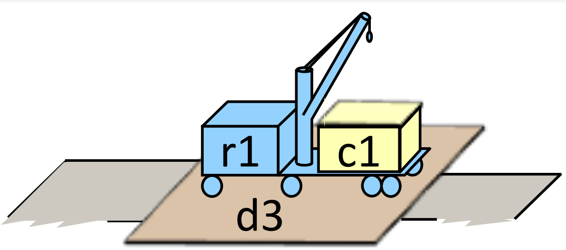
```
// extract minimal relaxed solution
ĝ_k = g
for i = k down to 1 do
    â_i = minimal subset of A_i s.t. γ⁺(ŝ_{i-1},â_i) r-satisfies ĝ_i
    ĝ_{i-1} ← (ĝ_i \ eff(â_i)) ∪ pre(â_i)
```

Could have followed other eff. line (would have lead to $h^{FF}(s_1) = 2$)

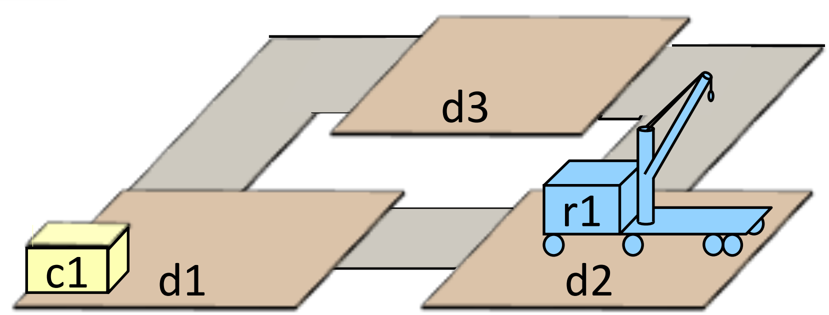
RPG from $\hat{s}_0 = s_2$ to g

Atoms in $\hat{s}_0 = s_2$:	Actions in A_1:	Atoms in $\hat{s}_1$:	Actions in A_2:	Atoms in $\hat{s}_2$:
$loc(r1) = d2$	$move(r1, d2, d3)$	$loc(r1) = d3$	$move(r1, d3, d2)$	$loc(r1) = d2$
$loc(c1) = d1$	$\boxed{move(r1, d2, d1)}$	$loc(r1) = d1$	$move(r1, d1, d2)$	$loc(c1) = d1$
$cargo(r1) = nil$	$\langle \hat{a}_1 \rangle$	$loc(r1) = d2$	$move(r1, d3, d1)$	$cargo(r1) = nil$
	from $\hat{s}_0$:	$loc(c1) = d1$	$\boxed{move(r1, d1, d3)}$	$loc(r1) = d1$
		$cargo(r1) = nil$	$move(r1, d2, d1)$	$loc(r1) = d3$
			$move(r1, d2, d3)$	$cargo(r1) = c1$
			$\boxed{take(r1, c1, d1)}$	$loc(c1) = r1$
			$\langle \hat{a}_2 \rangle$	from $\hat{s}_1$

- $\langle \hat{a}_1, \hat{a}_2 \rangle$ is a minimal relaxed solution
- Cost of each action is 1, so $h^{FF}(s_1) = 3$



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



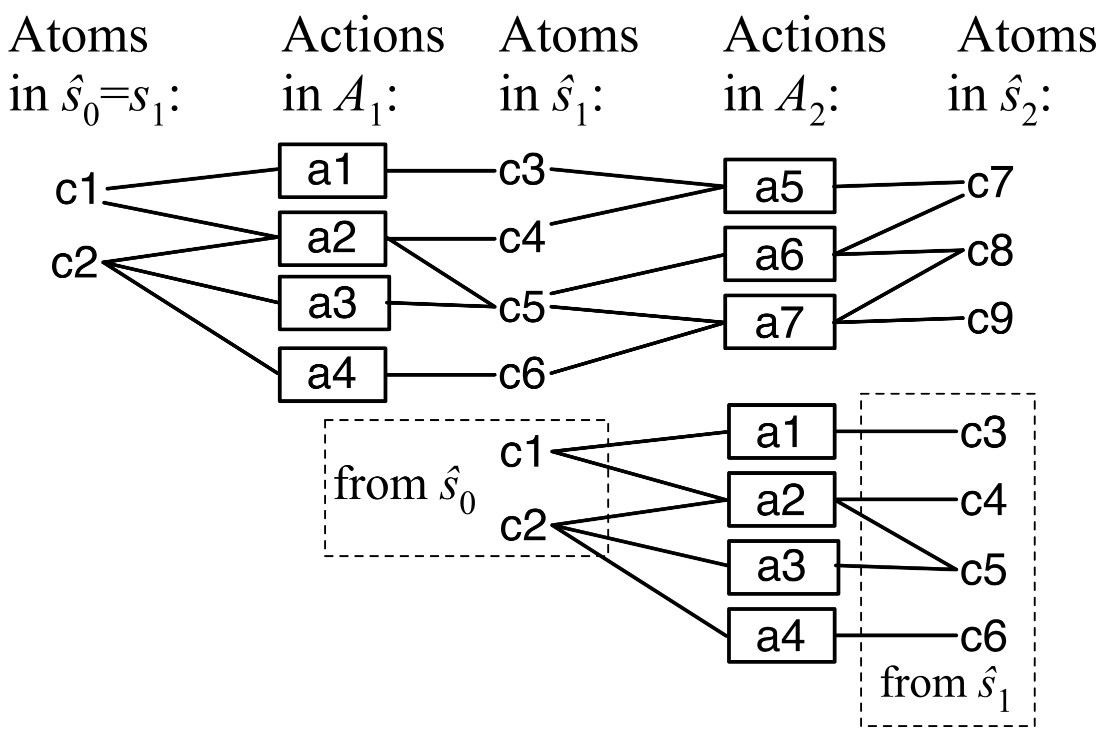
$s_2 = \{loc(r1) = d2, cargo(r1) = nil,\ loc(c1) = d1\}$

Properties

- Running time is polynomial in $|A| + \sum_{x \in X} |\mathcal{R}(x)|$
- *Minimal solution* doesn't mean *smallest cost*
  - A solution π to P is minimal if
    - no subsequence of π is also a solution for P.
  - A solution π to P is shortest if
    - there is no solution π' such that $|\pi'| < |\pi|$.
  - A solution π to P is cost-optimal if
    - $cost(\pi) = \min\{cost(\pi') | \pi' \text{is a solution for } P\}$.
  - $h^{FF}(s)$ = value returned by HFF(Σ, s, g)
    - $h^{FF}(s)$ = ∑ costs of $\hat{a}_1$, …, $\hat{a}_k$
  - $h^{FF}(s) \geq h^{+}(s)$ = *smallest* cost of any relaxed plan from s to goal
  - h^{FF} not admissible

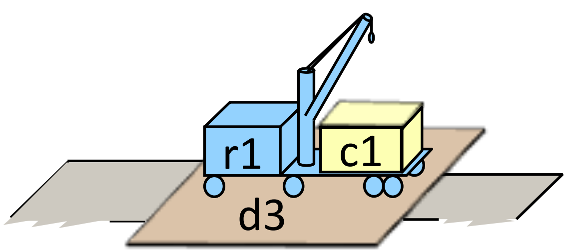
Example

- Suppose the goal atoms are c7, c8, c9. How many minimal solutions are there?
  - Assume default cost of 1

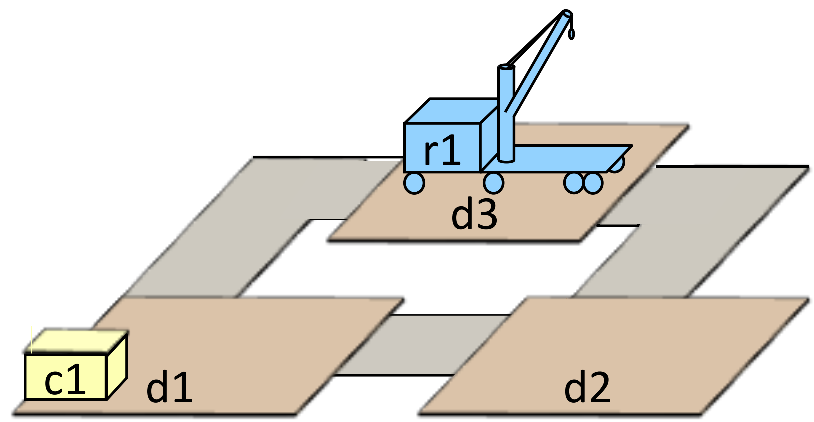


Landmark Heuristics

- $P = (\Sigma, s_0, g)$ be a planning problem
- Let $\varphi = \varphi_1 \vee \cdots \vee \varphi_m$ be a disjunction of ground atoms
- φ is a landmark for P if φ is true at some point in every solution for P
- Example landmarks
  - $loc(r1) = d1$
  - $loc(r1) = d3 \vee loc(r1) = d2$
  - $loc(r1) = d3$



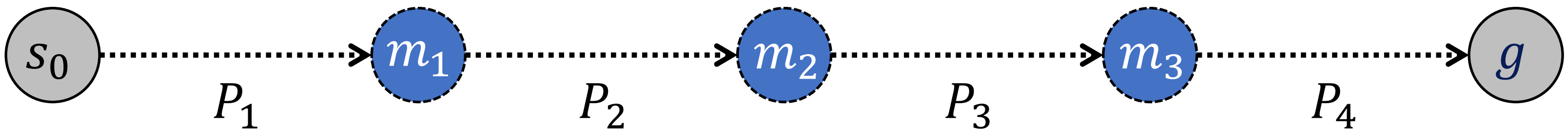
$g = \{loc(r1) = d3, loc(c1) = r1\}$



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

Why are Landmarks Useful?

- Breaks down a problem into smaller subproblems



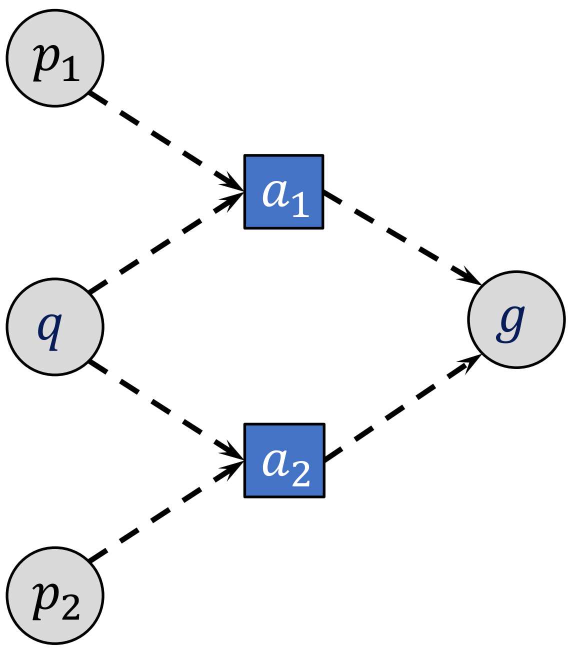
- Suppose m_1, m_2, m_3 are landmarks
  - Every solution to P must achieve m_1, m_2, m_3
- Possible strategy:
  - find a plan to go from s_0 to any state s_1 that satisfies m_1
  - find a plan to go from s_1 to any state s_2 that satisfies m_2
  - …

Computing Landmarks

- Worst-case complexity:
  - Deciding whether φ is a landmark is PSPACE-hard
  - As hard as solving the planning problem itself
- But there are often useful landmarks that can be found more easily
  - Polynomial time
  - Going to see one such procedure based on *RPGs*
    - Why RPGs?
      - Solving relaxed planning problems easier
        - Computing landmarks for relaxed planning problems easier
      - A landmark for a relaxed planning problem is a landmark for the original planning problem as well

RPG-based Landmark Computation

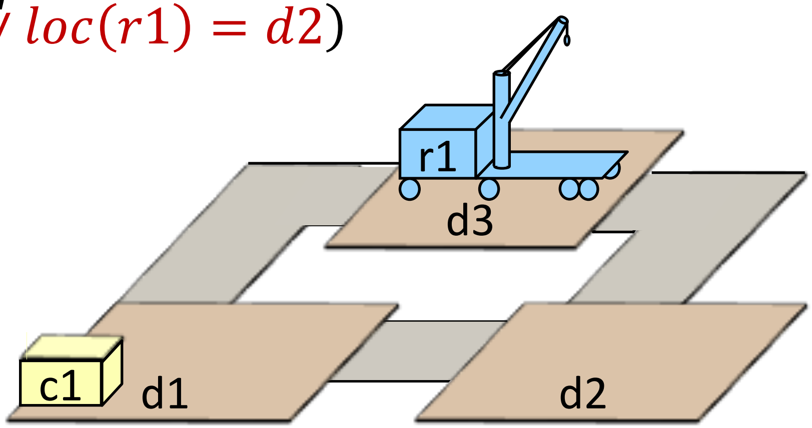
- Main intuition:
  - if φ is a landmark, can get new landmarks from the preconditions of the actions that achieve φ
- Example:
  - goal g
  - $\{a_1, a_2\}$ = all actions that achieve g
  - $pre(a_1) = \{p_1, q\}$
  - $pre(a_2) = \{q, p_2\}$
  - To achieve g, must achieve $(p_1 \wedge q) \vee (p_2 \wedge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



RPG-based Landmark Computation

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

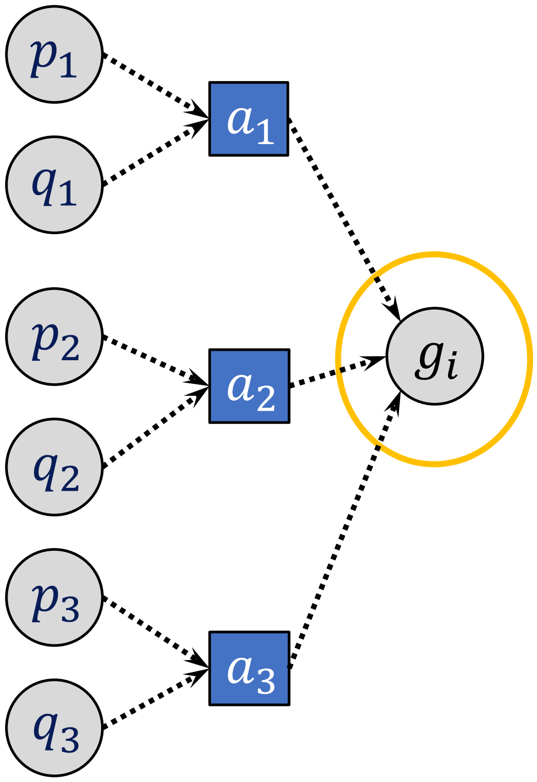
- $move(r, l, m)$
  - pre: $loc(r) = l$
  - eff: $loc(r) \leftarrow m$
- $take(r, l, c)$
  - pre: $cargo(r) = nil$, $loc(r) = l, loc(c) = l$
  - eff: $cargo(r) \leftarrow c, loc(c) \leftarrow r$
- $put(r, l, c)$
  - pre: $loc(r) = l, loc(c) = r$
  - eff: $cargo(r) \leftarrow nil, loc(c) \leftarrow l$



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

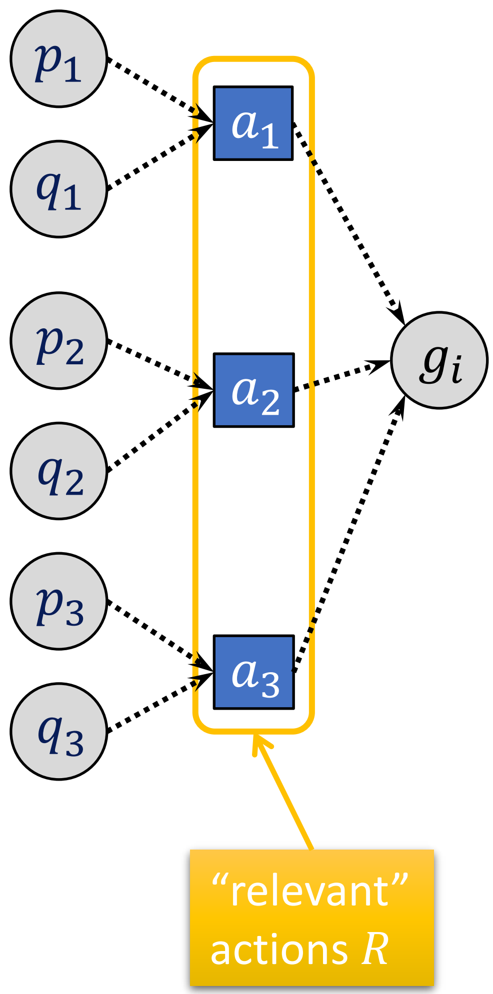
RPG-based Landmark Computation

```
RPG-Landmarks(s₀, g = {g₁, g₂,…, gₖ})
    queue ← {gᵢ ∈ g | s₀ doesn't satisfy gᵢ};
    Landmarks ← ∅
    A ← all actions
    while queue ≠ ∅ do
        remove a gᵢ from queue
        Landmarks ← Landmarks ∪ gᵢ
        R ← {actions whose effects include gᵢ}
        if s₀ satisfies pre(a) for some a ∈ R then
            return Landmarks
        generate RPG from s₀ and A \ R, stop when ŝₖ=ŝₖ₋₁
        N ← {a ∈ R | a r-applicable in ŝₖ}
        if N = ∅ then
            return failure
        Pre ← ∪{pre(a)|a ∈ N}\ s₀
        Φ ← {p₁∨p₂∨…∨pₘ|m≤4,∀a∈N ∃i:pᵢ∈pre(a),∀i:pᵢ∈Pre}
        for each φ ∈ Φ do
            add φ to queue
    return Landmarks
```



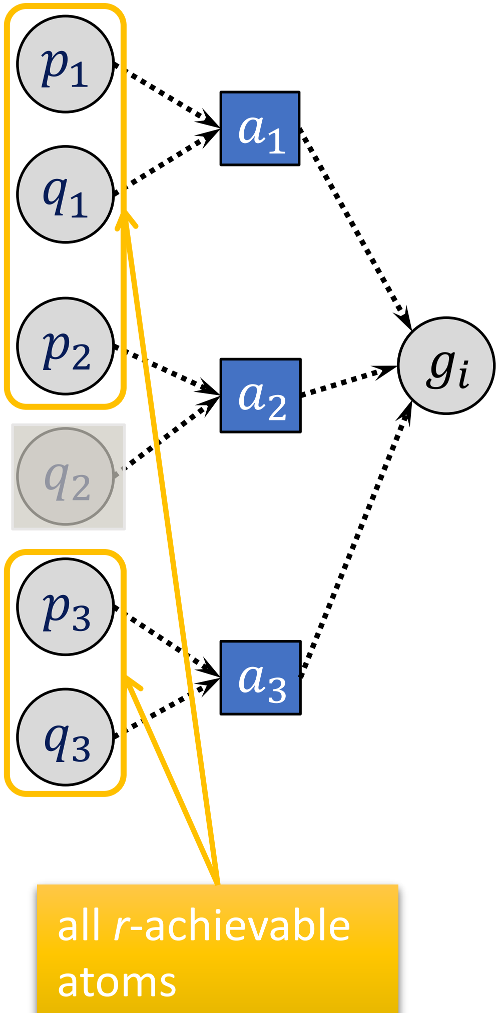
RPG-based Landmark Computation

```
RPG-Landmarks(s₀, g = {g₁, g₂,…, gₖ})
    queue ← {gᵢ ∈ g | s₀ doesn't satisfy gᵢ};
    Landmarks ← ∅
    A ← all actions
    while queue ≠ ∅ do
        remove a gᵢ from queue
        Landmarks ← Landmarks ∪ gᵢ
        R ← {actions whose effects include gᵢ}
        if s₀ satisfies pre(a) for some a ∈ R then
            return Landmarks
        generate RPG from s₀ and A \ R, stop when ŝₖ=ŝₖ₋₁
        N ← {a ∈ R | a r-applicable in ŝₖ}
        if N = ∅ then
            return failure
        Pre ← ∪{pre(a)|a ∈ N}\ s₀
        Φ ← {p₁∨p₂∨…∨pₘ|m≤4,∀a∈N ∃i:pᵢ∈pre(a),∀i:pᵢ∈Pre}
        for each φ ∈ Φ do
            add φ to queue
    return Landmarks
```



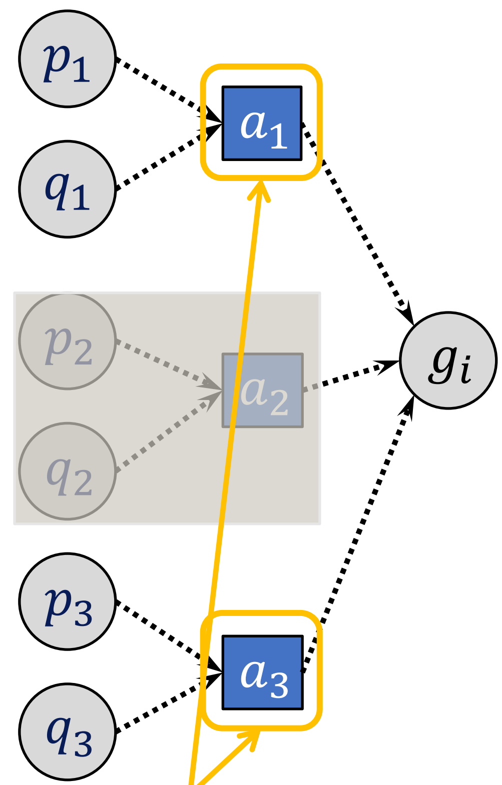
RPG-based Landmark Computation

```
RPG-Landmarks(s₀, g = {g₁, g₂,…, gₖ})
    queue ← {gᵢ ∈ g | s₀ doesn't satisfy gᵢ};
    Landmarks ← ∅
    A ← all actions
    while queue ≠ ∅ do
        remove a gᵢ from queue
        Landmarks ← Landmarks ∪ gᵢ
        R ← {actions whose effects include gᵢ}
        if s₀ satisfies pre(a) for some a ∈ R then
            return Landmarks
        generate RPG from s₀ and A \ R, stop when ŝₖ=ŝₖ₋₁
        N ← {a ∈ R | a r-applicable in ŝₖ}
        if N = ∅ then
            return failure
        Pre ← ∪{pre(a)|a ∈ N}\ s₀
        Φ ← {p₁∨p₂∨…∨pₘ|m≤4,∀a∈N ∃i:pᵢ∈pre(a),∀i:pᵢ∈Pre}
        for each φ ∈ Φ do
            add φ to queue
    return Landmarks
```



RPG-based Landmark Computation

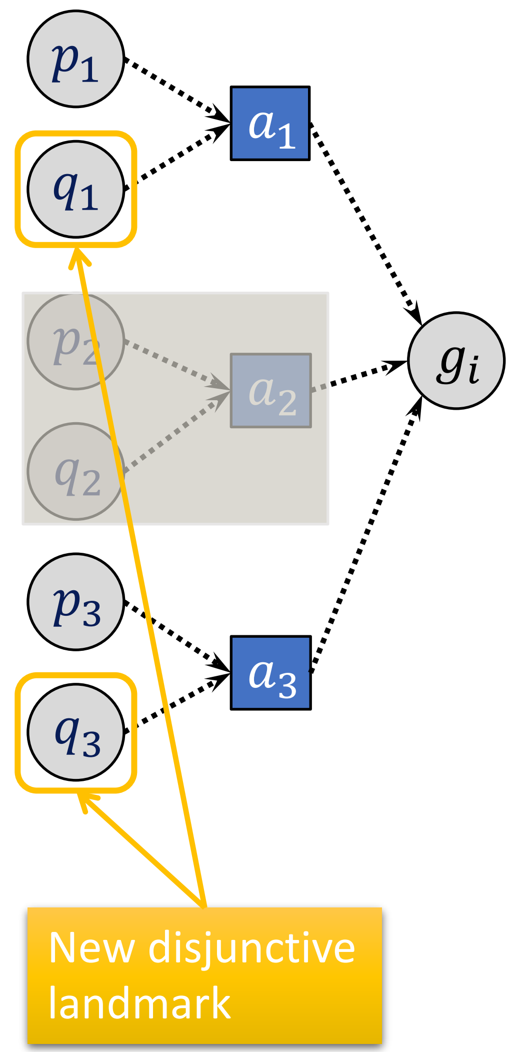
```
RPG-Landmarks(s0, g = {g1, g2,…, gk})
    queue ← {gi ∈ g | s0 doesn't satisfy gi};
    Landmarks ← ∅
    A ← all actions
    while queue ≠ ∅ do
        remove a gi from queue
        Landmarks ← Landmarks ∪ gi
        R ← {actions whose effects include gi}
        if s0 satisfies pre(a) for some a ∈ R then
            return Landmarks
        generate RPG from s0 and A \ R, stop when ŝk=ŝk-1
        N ← {a ∈ R | a r-applicable in ŝk}
        if N = ∅ then
            return failure
        Pre ← ∪{pre(a)|a ∈ N}\ s0
        Φ ← {p1∨p2∨…∨pm|m≤4,∀a∈N ∃i:pi∈pre(a),∀i:pi∈Pre}
        for each φ ∈ Φ do
            add φ to queue
    return Landmarks
```



"necessary" actions N: the only ones that can be r-applied and achieve g_i

RPG-based Landmark Computation

```
RPG-Landmarks(s0, g = {g1, g2,…, gk})
    queue ← {gi ∈ g | s0 doesn't satisfy gi};
    Landmarks ← ∅
    A ← all actions
    while queue ≠ ∅ do
        remove a gi from queue
        Landmarks ← Landmarks ∪ gi
        R ← {actions whose effects include gi}
        if s0 satisfies pre(a) for some a ∈ R then
            return Landmarks
        generate RPG from s0 and A \ R, stop when ŝk=ŝk-1
        N ← {a ∈ R | a r-applicable in ŝk}
        if N = ∅ then
            return failure
        Pre ← ∪{pre(a)|a ∈ N}\ s0
        Φ ← {p1∨p2∨…∨pm|m≤4,∀a∈N ∃i:pi∈pre(a),∀i:pi∈Pre}
        for each φ ∈ Φ do
            add φ to queue
    return Landmarks
```



Example

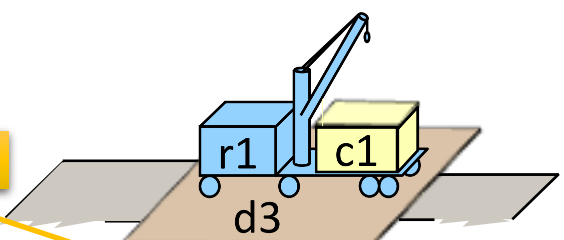
```
RPG-Landmarks(s₀, g = {g₁, g₂,…, gₖ})
    queue ← {gᵢ ∈ g | s₀ doesn't satisfy gᵢ};
    Landmarks ← ∅
    A ← all actions
    while queue ≠ ∅ do
        ...
```

- $move(r,l,m)$
  - pre: $loc(r) = l$
  - 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: $cargo(r) \leftarrow nil$, $loc(c) \leftarrow l$

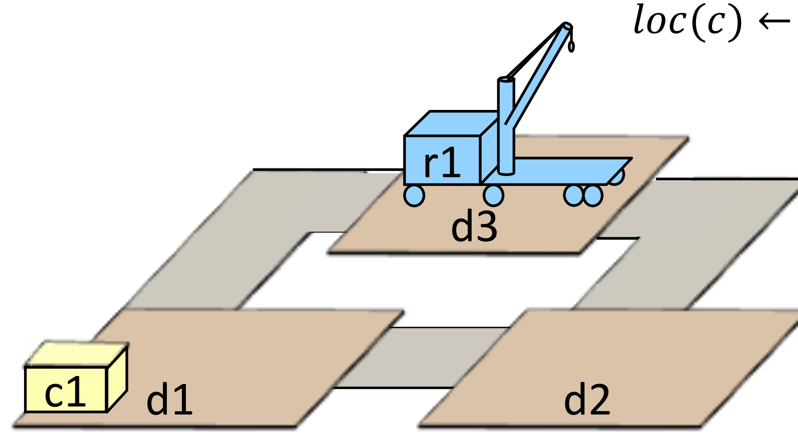
$queue = \{loc(c1) = r1\}$

$Landmarks = \emptyset$

true in s_0



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



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

Example

```
RPG-Landmarks(s_0, g = {g_1, g_2,…, g_k})
    ...
    while queue ≠ ∅ do
        remove a g_i from queue
        Landmarks ← Landmarks ∪ g_i
        R ← {actions whose effects include g_i}
        if s_0 satisfies pre(a) for some a ∈ R then
            return Landmarks
```

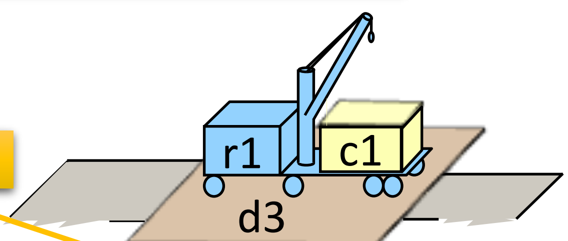
- $move(r, l, m)$
  - pre: $loc(r) = l$
  - eff: $loc(r) \leftarrow m$
- $take(r, l, c)$
  - pre: $cargo(r) = nil,$ $loc(r) = l, loc(c) = l$
  - eff: $cargo(r) \leftarrow c,$ $loc(c) \leftarrow r$
- $put(r, l, c)$
  - pre: $loc(r) = l,$ $loc(c) = r$
  - eff: $cargo(r) \leftarrow nil,$ $loc(c) \leftarrow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

$queue = \emptyset$

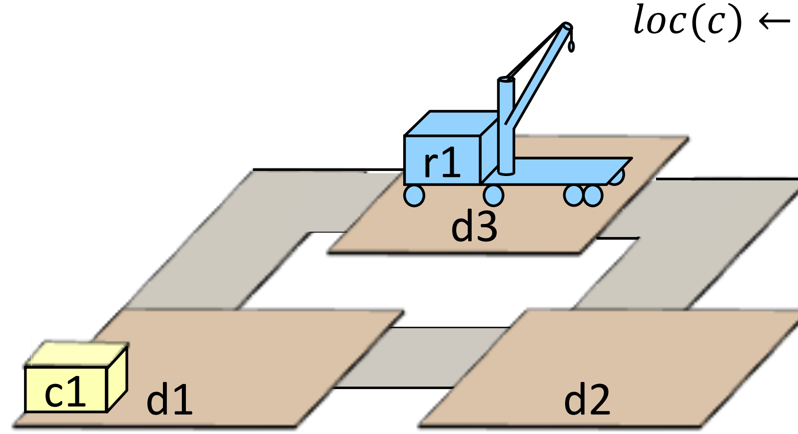
$Landmarks = \{loc(c1) = r1\}$

$R = \{take(r1, d1, c1),$
$take(r1, d2, c1),$
$take(r1, d3, c1)\}$

true in s_0



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



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

Example

RPG-Landmarks(s_0, $g = \{g_1, g_2, \ldots, g_k\}$)
  ...
  while $queue \neq \emptyset$ **do**
    ...
    generate RPG from s_0 and $A \setminus R$, stop when $\hat{s}_k = \hat{s}_{k-1}$
    $N \leftarrow \{a \in R \mid a$ r-applicable in $\hat{s}_k\}$
    if $N = \emptyset$ **then**
      return failure

- $move(r, l, m)$
  - pre: $loc(r) = l$
  - 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: $cargo(r) \leftarrow nil$, $loc(c) \leftarrow l$

$queue = \emptyset$

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

$R = \{take(r1, d1, c1),$
$\quad take(r1, d2, c1),$
$\quad take(r1, d3, c1)\}$

$N = \{take(r1, d1, c1)\}$

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

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

both $\hat{s}_1$ and $\hat{s}_2$:
$loc(r1) = d1$
$loc(r1) = d2$
From $\hat{s}_0$ $\begin{cases} loc(c1) = d1 \\ loc(r1) = d3 \\ cargo(r1) = nil \end{cases}$

$take(r1, c1, d1)$

RPG using $A \setminus R$

Example

```
RPG-Landmarks(s₀, g = {g₁, g₂,…, gₖ})
    ...
    while queue ≠ ∅ do
        ...
        Pre ← ⋃{pre(a) | a ∈ N}\ s₀
        Φ ← {p₁∨p₂∨…∨pₘ | m≤4, ∀a∈N ∃i:pᵢ∈pre(a), ∀i:pᵢ∈Pre}
        for each φ ∈ Φ do
            add φ to queue
```

- $move(r, l, m)$
  - pre: $loc(r) = l$
  - 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: $cargo(r) \leftarrow nil$, $loc(c) \leftarrow l$

$queue = \{loc(c1) = d1\}$

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

$R = \{take(r1, d1, c1), take(r1, d2, c1), take(r1, d3, c1)\}$

$N = \{take(r1, d1, c1)\}$

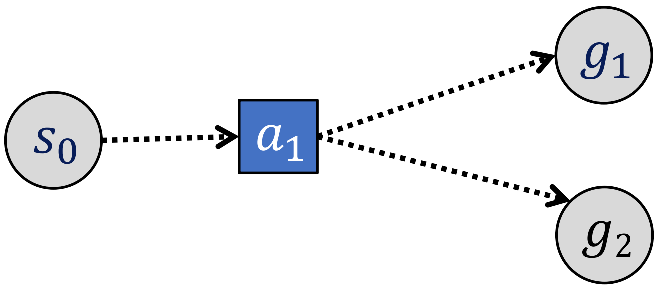
$take(r1, d1, c1)$
pre: $cargo(r1) = nil$,
$loc(r1) = d1$,
$loc(c1) = d1$

satisfied in $\hat{s}_0$

add to *queue*

Landmark Heuristic

- Every solution to the problem needs to achieve all the computed landmarks
- One possible heuristic:
  - $h^{sl}(s)$ = number of landmarks returned by RPG-Landmarks
  - Is this heuristic admissible?
    - No



$g = \{g_1, g_2\}$

Two landmarks: g_1, g_2

Optimal plan: $\langle a_1 \rangle$, length = 1

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

Intermediate Summary

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

Outline per the Book

2.1 *State-variable representation*

- State = {values of variables}; action = changes to those values

2.2 *Forward state-space search*

- Start at initial state, look for sequence of actions that achieve goal

2.3 *Heuristic functions*

- How to guide a forward state-space search

2.6 *Incorporating planning into an actor*

- Online lookahead, unexpected events

2.4 *Backward search*

- Start at goal state, go backwards toward initial state

2.5 *Plan-space search*

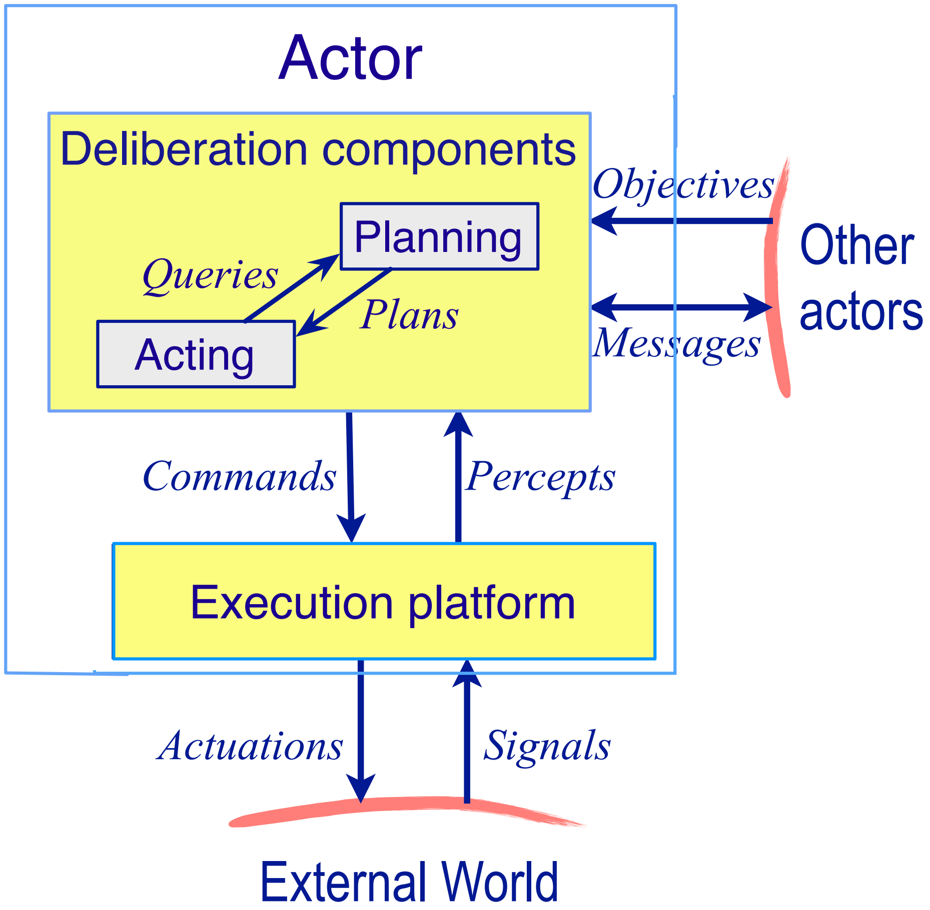
- Start with incomplete plan for getting from initial state to goal state, make transformations to fix flaws in the plan

Incorporating Planning into an Actor

- Plans are abstract
  - Need additional refinement
  - (Chapter 3)

The best laid schemes o' mice an' men,
Gang aft agley.
–Robert Burns

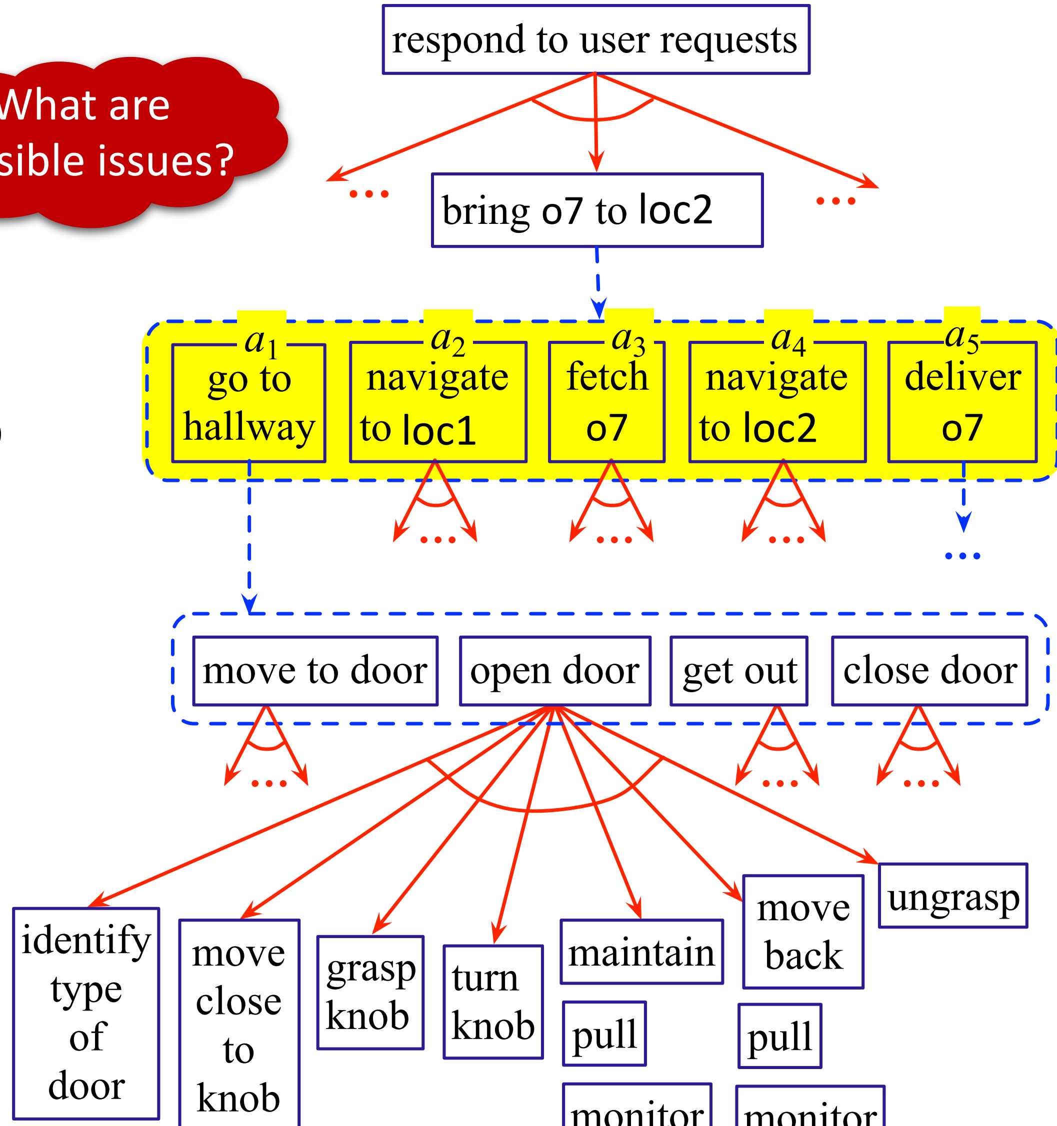
- Plans don't always work
  - What to do about it?



Service Robot

- $s_0 = \{loc(r1) = loc3, loc(o7) = loc1, cargo(r1) = nil\}$
- $g = \{loc(o7) = loc2\}$
- $\pi = \langle a1, a2, a3, a4, a5 \rangle$
  - $a1 = go(r1, loc3, hall)$
  - $a2 = navigate(r1, hall, loc1)$
  - $a3 = take(r1, loc1, o7)$
  - $a4 = navigate(r1, loc1, loc2)$
  - $a5 = put(r1, loc2, o7)$
- $go(r, l, m)$
  - pre: $adj(l, m), loc(r) = l$
  - eff: $loc(r) \leftarrow m$
- $navigate(r, l, m)$
  - pre: $\neg adj(l, m), loc(r) = l$
  - eff: $loc(r) \leftarrow m$
- $take(r, l, o)$
  - pre: $loc(r) = l, loc(o) = l, cargo(r) = nil$
  - eff: $loc(o) \leftarrow r, cargo(r) \leftarrow o$

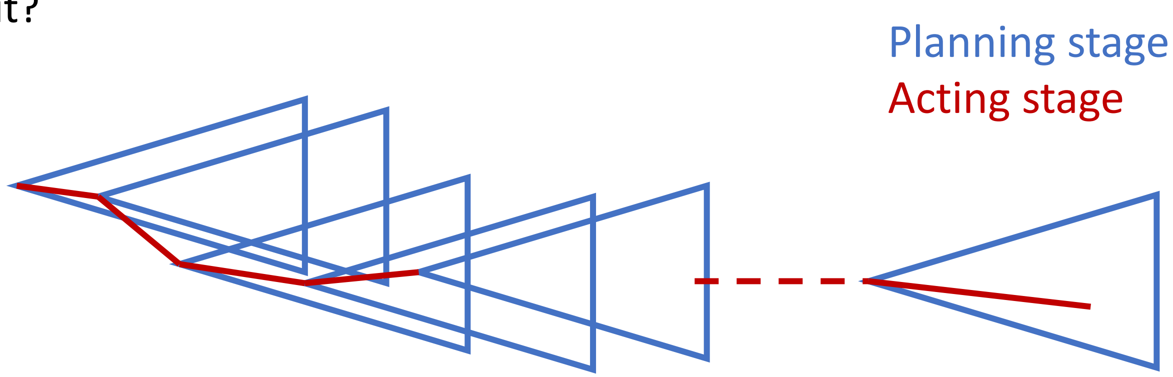
What are possible issues?



Using Planning in Acting

- Lookahead is the planner
- Receding horizon:
  - Call Lookahead, obtain π, perform 1st action, call Lookahead again …
  - Like game-tree search (chess, checkers, etc.)
- Useful when unpredictable things are likely to happen
  - Re-plans immediately
- Potential problem:
  - May pause repeatedly while waiting for Lookahead to return
  - What if ξ changes during the wait?

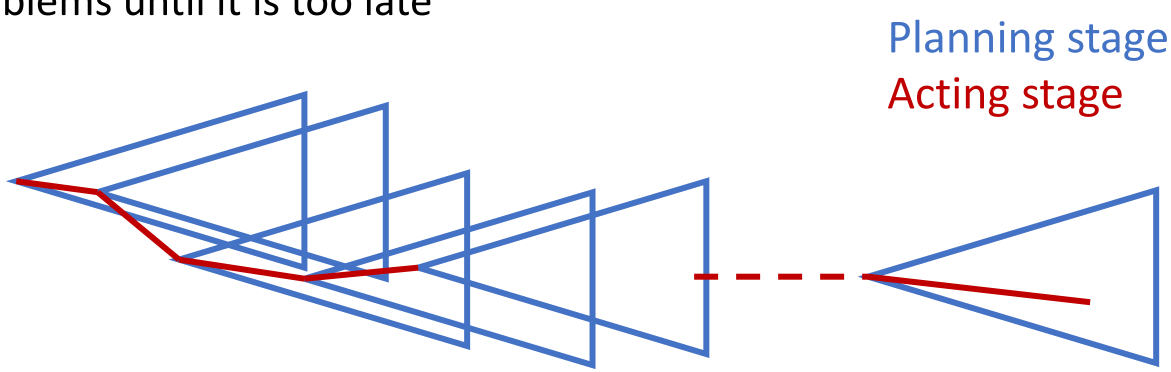
```
Run-Lookahead(Σ,g)
    while s ← abstraction of
              observed state ξ ⊭ g do
        π ← Lookahead(Σ,s,g)
        if π = failure then
            return failure
        a ← pop-first-action(π)
        perform a
```



Using Planning in Acting

- Call Lookahead, execute the plan as far as possible, don't call Lookahead again unless necessary
- Simulate tests whether the plan will execute correctly
  - Could just compute $\gamma(s, \pi)$, or could do something more detailed
    - Lower-level refinement, physics-based simulation
- Potential problems
  - May might miss opportunities to replace π with a better plan
  - Without Simulate, may not detect problems until it is too late

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



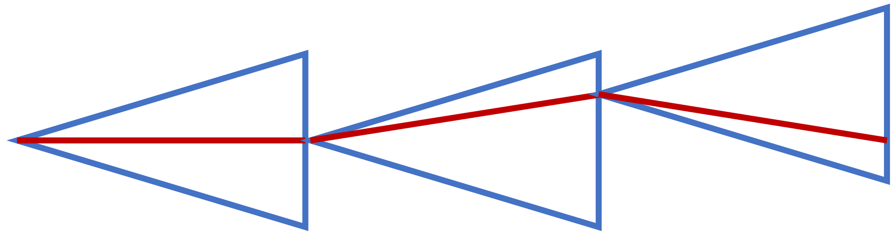
Using Planning in Acting

- May detect opportunities earlier than Run-Lazy-Lookahead
  - But may miss some that Run-Lookahead would find
- Without Simulate, may fail to detect problems until it is too late
  - Not as bad at this as Run-Lazy-Lookahead
  - Possible work-around: restart Lookahead each time s changes

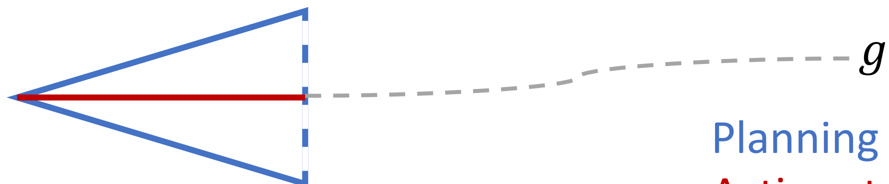
```
Run-Concurrent-Lookahead(Σ,g)
    π ← ⟨⟩
    s ← abstraction of
              observed state ξ
    // thread 1 + 2 run concurrently
thread 1:
    loop
        π ← Lookahead(Σ,s,g)
thread 2:
    loop
        if s ⊨ g then
            return success
        else if π = failure then
            return failure
        else if π ≠ ⟨⟩ and s ⊭ g and
              Simulate(Σ,s,g,π)
                  ≠ failure then
            a ← pop-first-action(π)
            perform a
            s ← abstraction of
                      observed state ξ
```

How to do Lookahead

- Subgoaling
  - Instead of planning for g, plan for a subgoal g'
  - Once g' is achieved, plan for next subgoal

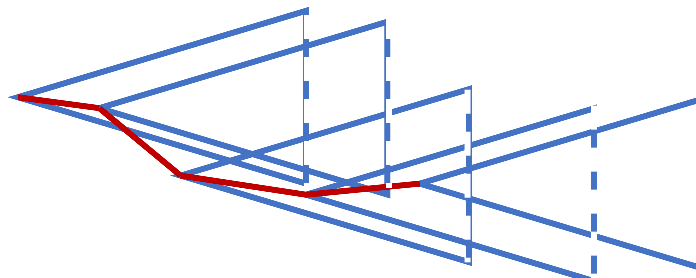


- Receding horizon
  - Return a plan that goes just part-way to g'
  - E.g., cut off search at
    - every plan whose cost exceeds some value c_{max}
    - or whose length exceeds some value l_{max}
    - or when no time is left
  - Horizon recedes on the actor's successive calls to the planner

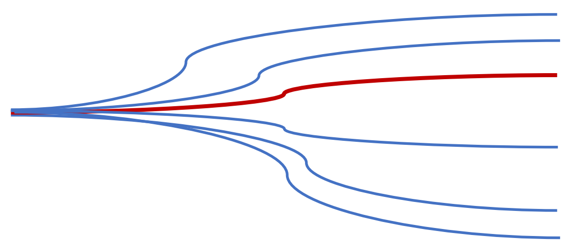


Planning stage

Acting stage



- Sampling
  - Try a few (e.g., randomly chosen) depth-first rollouts, take the one that looks best



- Can use combinations of these

Receding-Horizon Search

- After line (*i*), put something like these:
  - *cost-based cutoff*:

    if $cost(\pi) + h(s) > c_{max}$ then
    return π

  - *length-based cutoff* :

    if $|\pi| > l_{max}$ then
    return π

  - *time-based cutoff*:

    if time-left() = 0 then
    return π

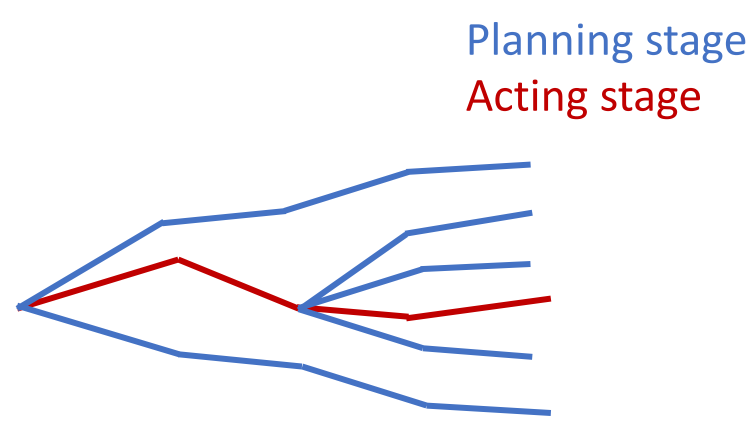
```
Deterministic-Search(Σ,s₀,g)
  Frontier ← {(⟨⟩,s₀)}
  Expanded ← ∅
  while Frontier ≠ ∅ do
    select a node ν = (π,s) ∈ Frontier  (i)
    remove ν from Frontier
    add ν to Expanded
    if s satisfies g then
      return π
    Children ←
      {(π.a, γ(s,a)) | s satisfies pre(a)}
    prune 0 or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

Partial or Non-optimal Plans

- Sampling
  - Planner is a modified version of greedy algorithm
  - Make randomized choice at (i)
  - Run several times, get several solutions
  - Return best one

```
Greedy(Σ,s₀,g,Visited)
    if s satisfies g then
        return π
    Act ← {a ∈ A| s satisfies pre(a) and
               γ(s,a) ∉ Visited}
    if Act = ∅ then
        return failure
    a ← argmin_{a∈Act} h(γ(s,a))            (i)
    π ← Greedy(Σ, γ(s,a),g,Visited ∪ {s})
    if π ≠ failure then
        return a.π
    return failure
```

- Actor calls the planner repeatedly as it acts
  - An analogous technique is used in the game of Go



Intermediate Summary

- Incorporating Planning into an actor
  - Things that can go wrong while acting
  - Algorithms
    - Run-Lookahead
    - Run-Lazy-Lookahead
    - Run-Concurrent-Lookahead
  - Lookahead
    - Subgoaling
    - Receding-horizon search
    - Sampling

Outline per the Book

2.1 *State-variable representation*

- State = {values of variables}; action = changes to those values

2.2 *Forward state-space search*

- Start at initial state, look for sequence of actions that achieve goal

2.3 *Heuristic functions*

- How to guide a forward state-space search

2.6 *Incorporating planning into an actor*

- Online lookahead, unexpected events

2.4 *Backward search*

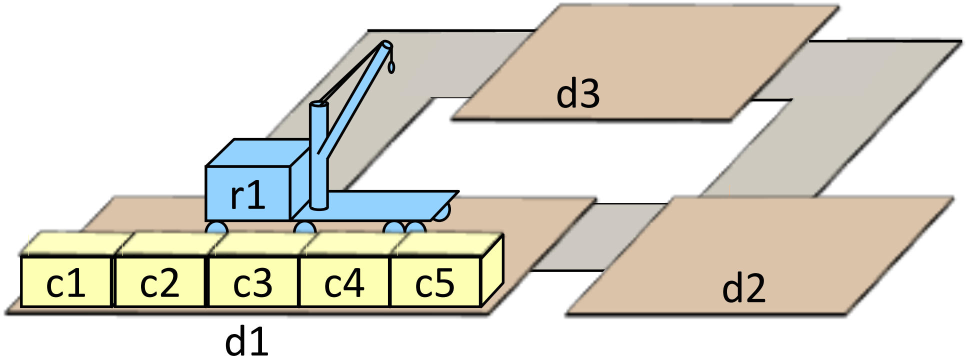
- Start at goal state, go backwards toward initial state

2.5 *Plan-space search*

- Start with incomplete plan for getting from initial state to goal state, make transformations to fix flaws in the plan

Backward Search

- Forward search starts at the initial state
  - Choose applicable action
  - Compute state transition $s' = \gamma(s, a)$
- Backward search starts at the goal
  - Chooses relevant action
    - A possible “last action” before the goal
  - Computes inverse state transition $g' = \gamma^{-1}(g, a)$
    - g' = properties a state s' should satisfy in order for $\gamma(s', a)$ to satisfy g
- Sometimes has a lower branching factor
  - Forward: 7 applicable actions
    - five load actions, two move actions
  - Backward: $g = \{loc(r1) = d3\}$
    - two relevant actions: $move(r1, d1, d3), move(r2, d1, d3)$



Relevance

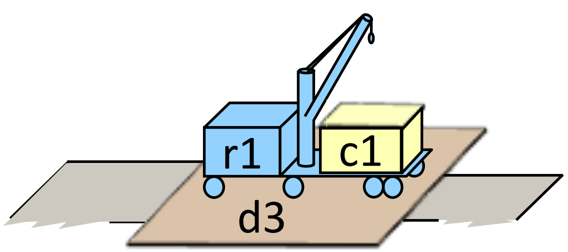
- Idea: when can a be useful as the last action of a plan π for achieving g?
  - a can make at least one atom in g true that wasn't true already
  - a doesn't make any part of g false

- a is relevant for $g = \{x_1 = c_1, \ldots, x_k = c_k\}$ if
  - at least one atom in g is also in $eff(a)$
    - i.e., g contains $x = c$ and $eff(a)$ contains $x \leftarrow c$
  - for every atom $x = c$ in g
    - a doesn't make $x = c$ false
      - i.e., $eff(a)$ doesn't contain $x \leftarrow c'$ for some $c' \neq c$
    - if $pre(a)$ requires $x = c$ to be false, then $eff(a)$ makes it true
      - i.e., if $pre(a)$ contains $x \neq c$ or $x = c'$, then $eff(a)$ contains $x \leftarrow c$

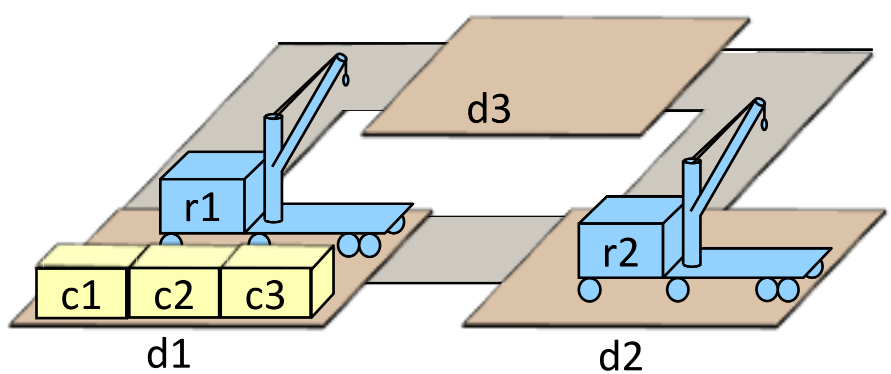
Relevance

- $adj = \{(d1, d2), (d1, d3), (d2, d1), (d2, d3), (d3, d1), (d3, d2)\}$
- $s = \{loc(c1) = d1, loc(c2) = d1, loc(c3) = d1, loc(r1) = d2, \ cargo(r1) = nil, loc(r2) = d2, \ cargo(r2) = nil\}$
- $g = \{loc(c1) = r1, loc(r1) = d3\}$
- For each action below, is it relevant for g?
  - $take(r1, d1, c1)$
  - $take(r1, d2, c1)$
  - $put(r2, d3, c1)$
  - $move(r1, d1, d3)$
  - $move(r1, d3, d1)$
  - $move(r1, d2, d3)$

- $move(r, l, m)$
  - pre: $loc(r) = l, adj(l, m)$
  - 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: $cargo(r) \leftarrow nil, loc(c) \leftarrow l$
- Ranges
  - $\mathcal{R}(r) = \ Robots \ = \{r1, r2\}$
  - $\mathcal{R}(l) = \mathcal{R}(m) = \ Locs \ = \{d1, d2, d3\}$
  - $\mathcal{R}(c) \ = \ Containers \ = \{c1, c2, c3\}$



g

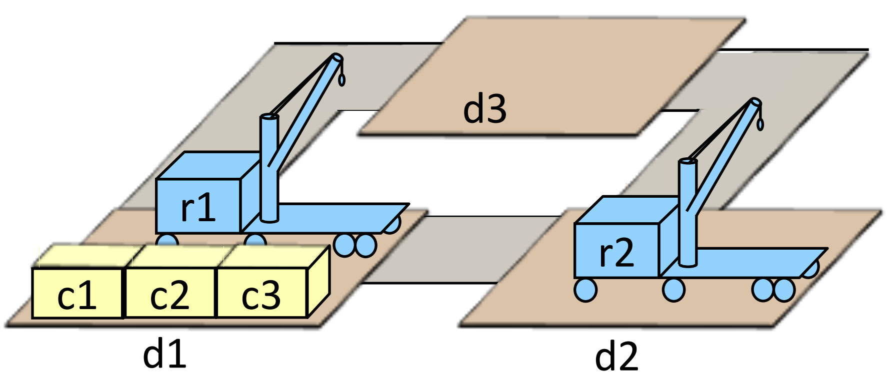


s

Inverse State Transitions

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

- $move(r,l,m)$
  - pre: $loc(r) = l, adj(l,m)$
  - 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: $cargo(r) \leftarrow nil, loc(c) \leftarrow l$



s

Backward Search

- Cycle checking:
  - After line (*i*), put
    $Solved \leftarrow \{g\}$
  - After line (*ii*), put
    - either this:

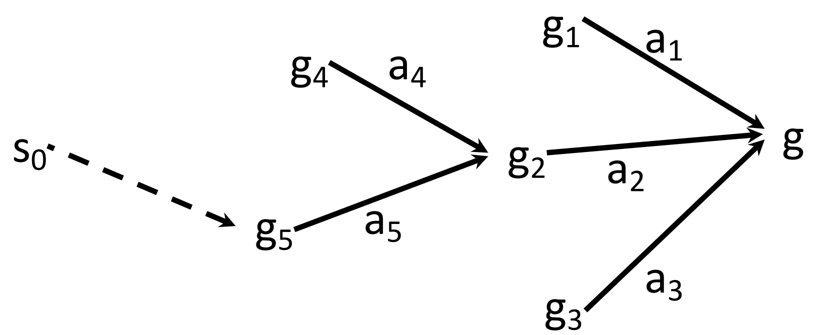
      if $g \in Solved$ then
      return failure
      $Solved \leftarrow Solved \cup \{g\}$

    - or this:

      if $\exists g' \in Solved$ s.t. $g \subseteq g'$ then
      return failure
      $Solved \leftarrow Solved \cup \{g\}$

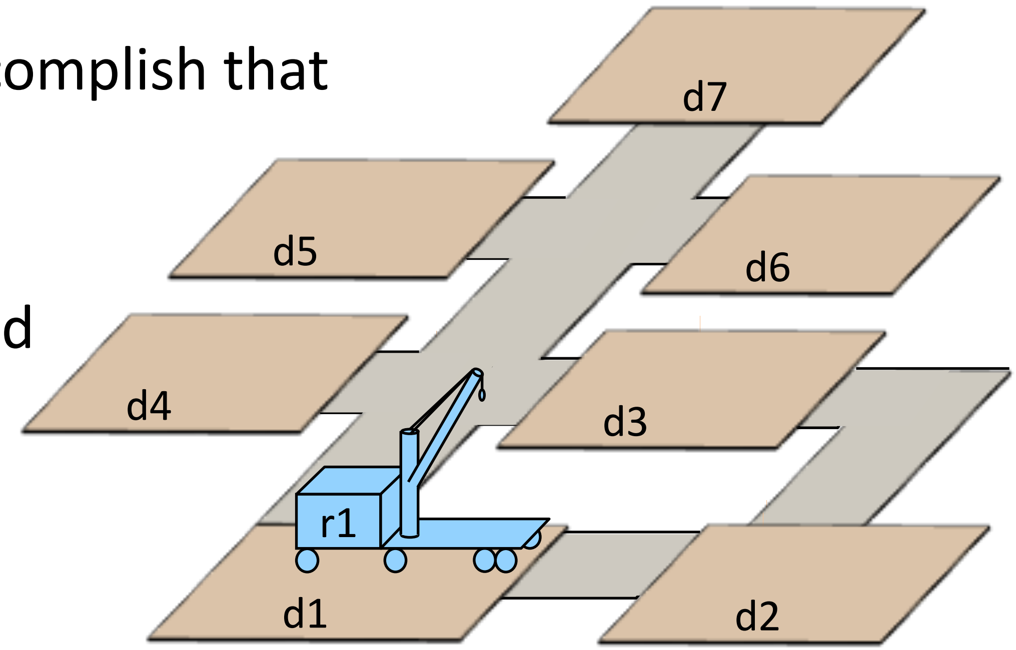
- With cycle checking, sound and complete
  - If (Σ, s_0, g_0) is solvable, then at least one of the execution traces will find a solution

```
Backward-search(Σ, s0, g0)
    g ← g0
    π ← ⟨⟩                                    (i)
    loop
        if s0 satisfies g then
            return π
        A' ←{a ∈ A | a is relevant for g}
        if A' = ∅ then
            return failure
        nondeterministically choose a ∈ A'
        g ← γ^-1(g, a)
        π ← a.π                                (ii)
```



Branching Factor

- Motivation for Backward-search was to reduce the branching factor
  - As written, doesn't accomplish that
- Solve this by lifting:
  - When possible, leave variables uninstantiated



$move(r1, d1, d3)$

$move(r1, d2, d3)$

$move(r1, d4, d3)$

$\vdots$

$move(r1, d7, d3)$

γ^{-1}

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

$move(r1, y, d3) \xleftarrow{\gamma^{-1}} g = \{loc(r1) = d3\}$

Lifted Backward Search

- Like Backward-search but much smaller branching factor
  - Must keep track of what values were substituted for which parameters
  - I won't discuss the details
  - Plan-space planning (later) does something similar

```
Backward-search(Σ,s₀,g₀)
    g ← g₀
    π ← ⟨⟩
    loop
        if s₀ satisfies g then
            return π
        A′ ←{a ∈ A | a is relevant for g}
        if A′ = ∅ then
            return failure
        nondeterministically choose a ∈ A′
        g ← γ⁻¹(g,a)
        π ← a.π
```

```
Lifted-Backward-search(𝒜,s₀,g)
    π ← ⟨⟩
    loop
        if s₀ satisfies g then
            return π
        A ← {(a,θ)| a is a standardisation of an action template in 𝒜,
                θ is an mgu for an atom of g and an atom of eff⁺(a), and
                  γ⁻¹(θ(g),θ(a)) is defined}
        if A = ∅ then
            return failure
        nondeterministically choose (a,θ) ∈ A
        g ← γ ⁻¹(θ(g),θ(a))
```

Intermediate Summary

- Backward State-space Search
  - Relevance, inverse state transition γ^{-1}
  - Backward search, cycle checking
  - Lifted backward search (briefly)

Outline per the Book

2.1 *State-variable representation*

- State = {values of variables}; action = changes to those values

2.2 *Forward state-space search*

- Start at initial state, look for sequence of actions that achieve goal

2.3 *Heuristic functions*

- How to guide a forward state-space search

2.6 *Incorporating planning into an actor*

- Online lookahead, unexpected events

2.4 *Backward search*

- Start at goal state, go backwards toward initial state

2.5 *Plan-space search*

- Start with incomplete plan for getting from initial state to goal state, make transformations to fix flaws in the plan

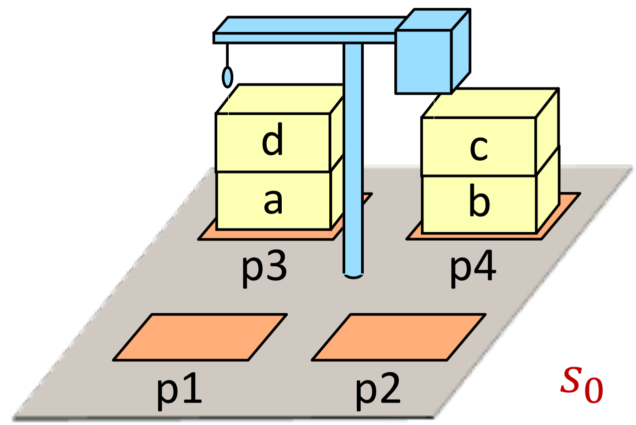
Plan-Space Search

- Formulate planning as a constraint satisfaction problem
  - Use constraint-satisfaction techniques to produce solutions that are more flexible than ordinary plans
    - E.g., plans in which the actions are partially ordered
    - Postpone ordering decisions until the plan is being executed
      - the actor may have a better idea about which ordering is best
- First step toward temporal planning (Chapter 4 in book)

- Basic idea:
  - Backward search from the goal
  - Each node of the search space is a partial plan that contains flaws
    - Remove the flaws by making refinements
  - If successful, we will get a partially ordered solution

Definitions

- Partially ordered plan
  - partially ordered set of nodes
  - each node contains an action
- Partially ordered solution
  - partially ordered plan π such that every total ordering of π is a solution
- Partial plan
  - partially ordered set of nodes that contain partially instantiated actions
  - inequality constraints
    - e.g. $z \neq x$ or $w \neq p1$
  - causal links (dashed arcs)
    - use action a to establish precondition p of action b



s_0

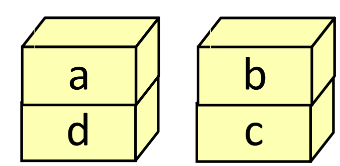
$move(d, a, p1) \rightarrow move(a, p3, d)$

$move(c, b, p4) \rightarrow move(b, p4, c)$

$foo(y) \rightarrow bar(y) \rightarrow baz(z)$

pre: $loc(y) = p1$ pre: $loc(z) = p2$

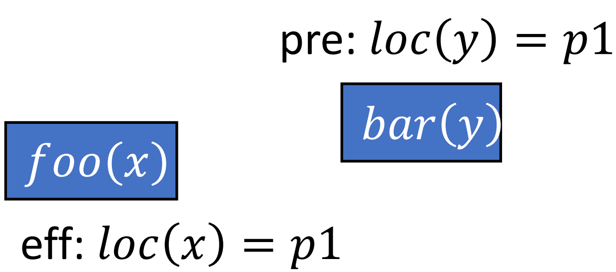
eff: $loc(y) = p1$ $z \neq x$

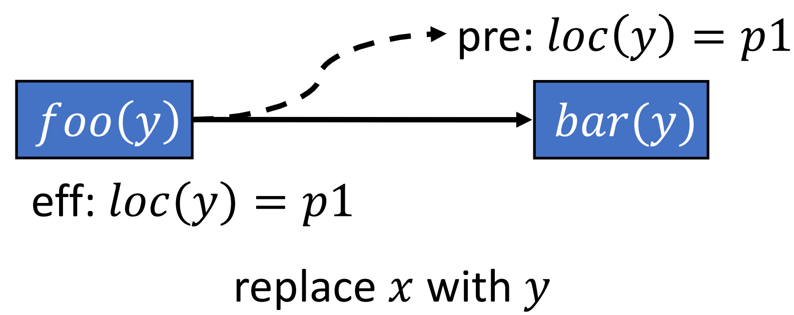


g

Flaws: 1. Open Goals

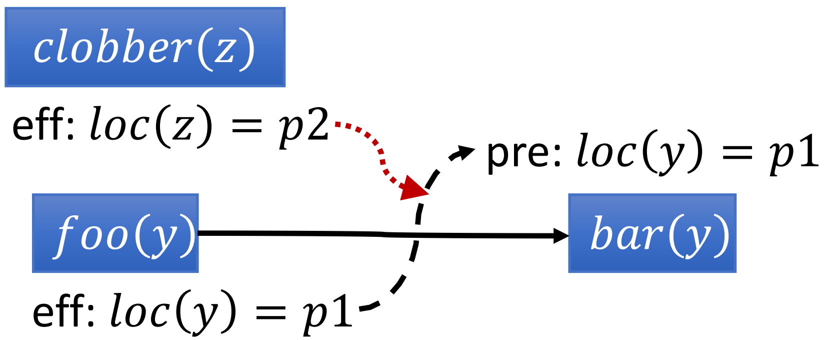
- A precondition p of an action b is an open goal if there is no causal link for p
- Resolve the flaw by creating a causal link
  - Find an action a (either already in π, or can add it to π) that can establish p
    - can precede b
    - can have p as an effect
  - Do substitutions on variables to make a assert p
    - e.g., replace x with y
  - Add an ordering constraint $a \prec b$
  - Create a causal link from a to p





Flaws: 2. Threats

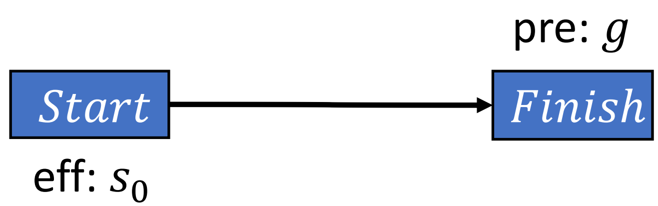
- Suppose we have a causal link from action a to precondition p of action b
- Action c threatens the link if c may affect p and may come between a and b
  - c is a threat even if it makes p true rather than false
    - Causal link means a, not c, is supposed to establish p for b
    - The plan in which c establishes p will be generated on another path in the search space
- Three possible ways to resolve the flaw:
  - Make $c \prec a$
  - Make $b \prec c$
  - Add inequality constraints to prevent c from affecting p



PSP Algorithm

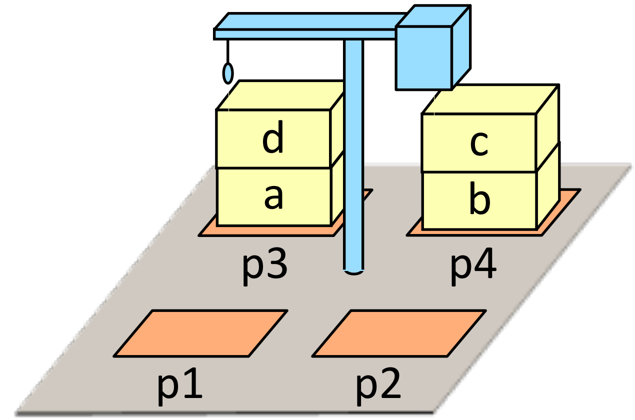
- Initial plan is always $\{Start, Finish\}$ with $Start \prec Finish$
  - Start
    - No preconditions
    - Effects: atoms in s_0
  - Finish
    - Preconditions: atoms in g
    - No effects
- PSP is sound and complete
  - Returns a partially ordered plan π s.t. any total ordering of π will achieve g
  - In some environments, could execute actions in parallel

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



Example

- Finish has two open goals: pos(a)=d, pos(b)=c



Start

clear(p1)=T clear(p2)=T clear(p3)=F clear(p4)=F
clear(a)=F clear(b)=F clear(c)=T clear(d)=T
pos(a)=p3 pos(b)=p4 pos(c)=b pos(d)=a

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

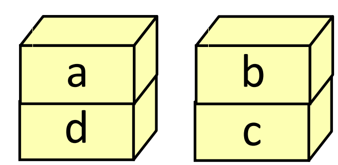
move(*c, y, z*)
pre: pos(*c*)=*y*, clear(*c*)=T, clear(*z*)=T
eff: pos(*c*)←*z*, clear(*y*)←T, clear(*z*)←F

pos(a)=d pos(b)=c

Finish

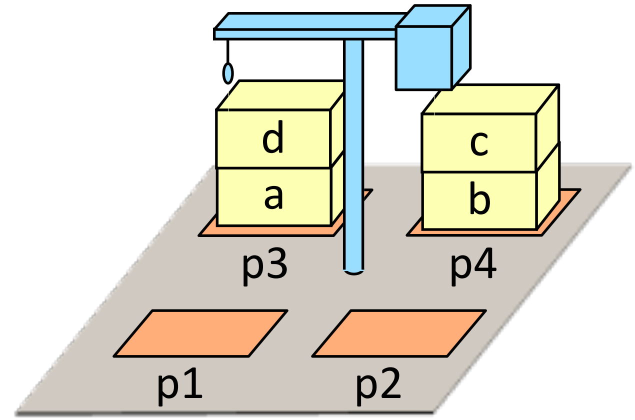
$\mathcal{R}(c)$ = *Containers*

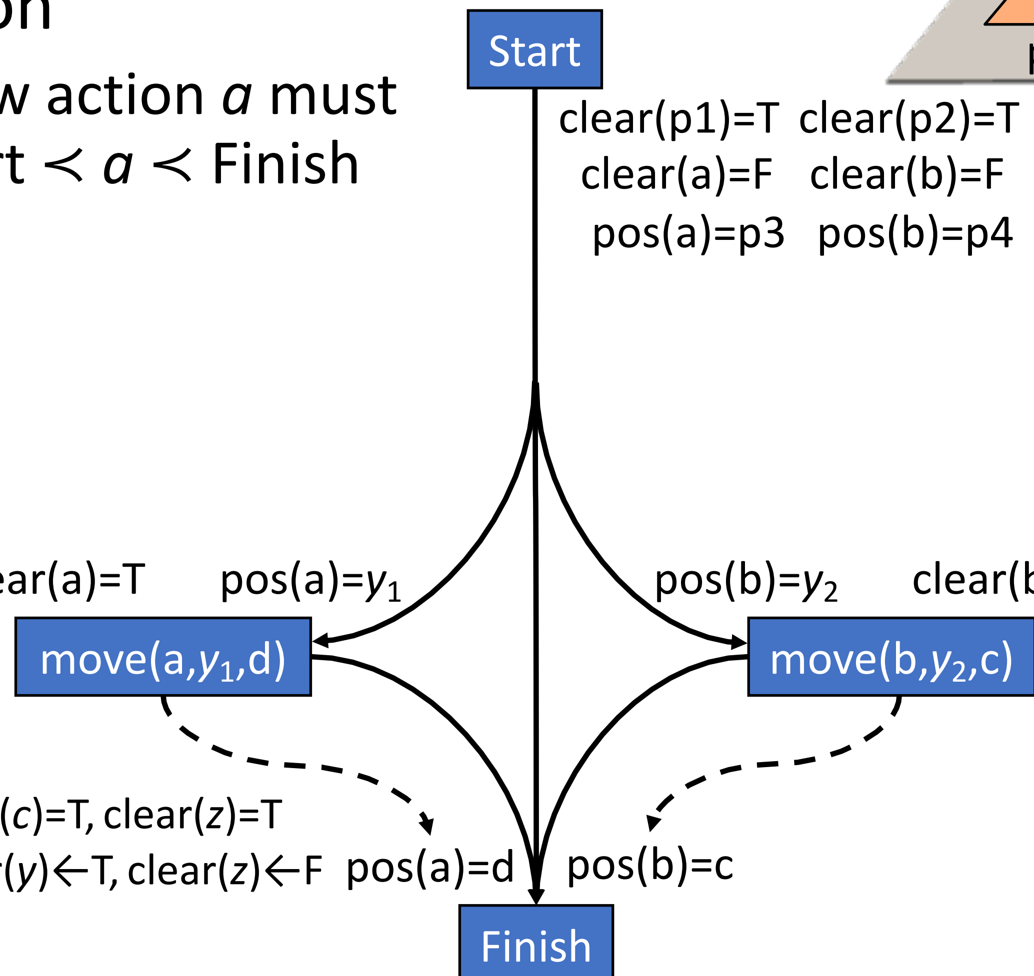
$\mathcal{R}(y) = \mathcal{R}(z)$ = *Container* ∪ *pallets*



Example

- For each open goal, add a new action
  - Every new action *a* must have Start $\prec a \prec$ Finish

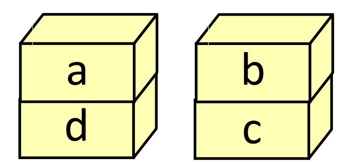




move(*c*, *y*, *z*)
pre: pos(*c*)=*y*, clear(*c*)=T, clear(*z*)=T
eff: pos(*c*)←*z*, clear(*y*)←T, clear(*z*)←F

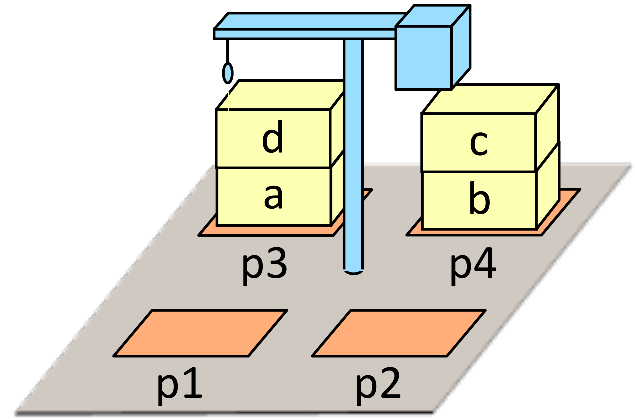
$\mathcal{R}(c)$ = *Containers*

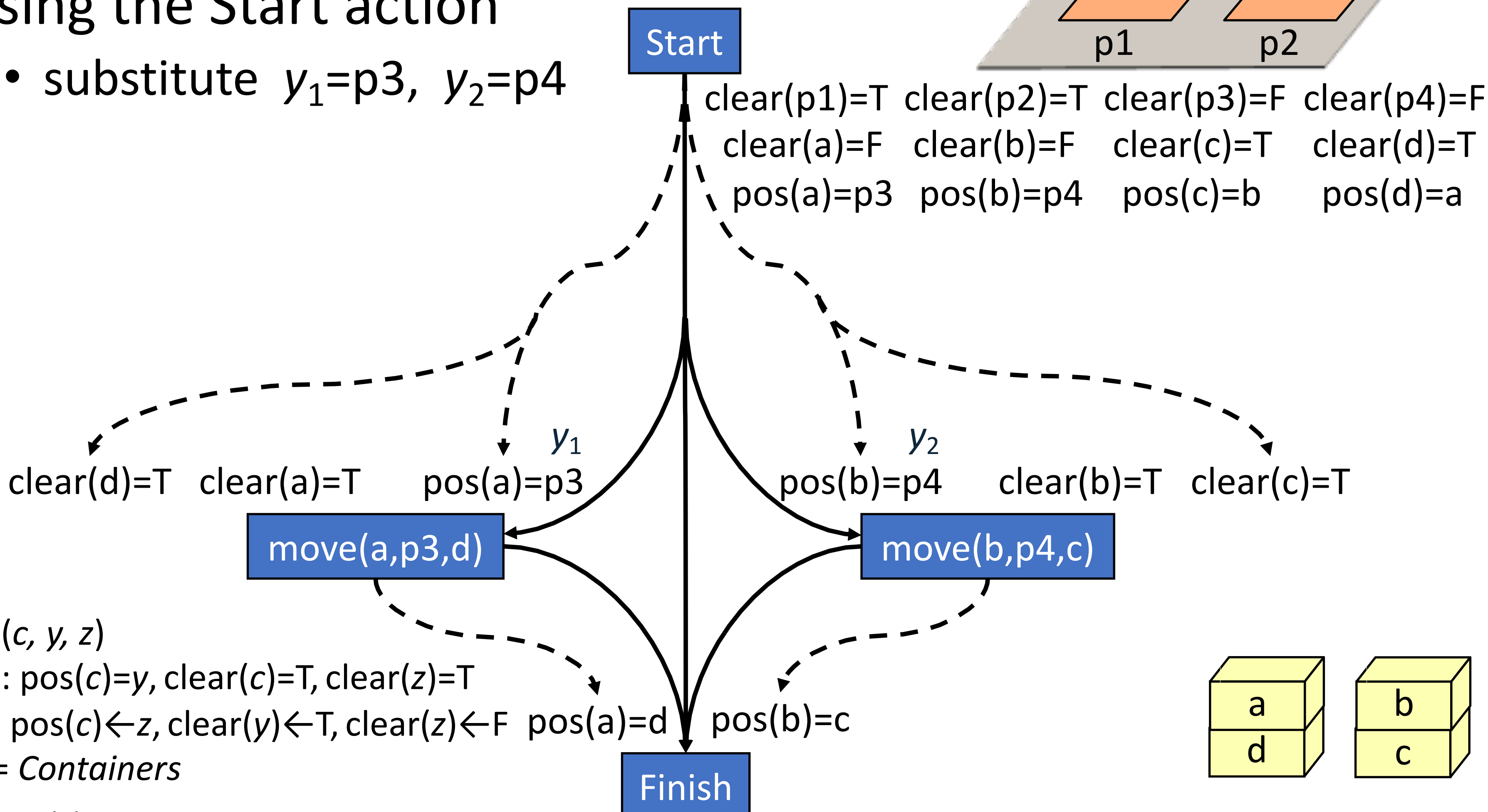
$\mathcal{R}(y) = \mathcal{R}(z)$ = *Container* ∪ *pallets*



Example

- Resolve four open goals using the Start action
  - substitute y_1=p3, y_2=p4





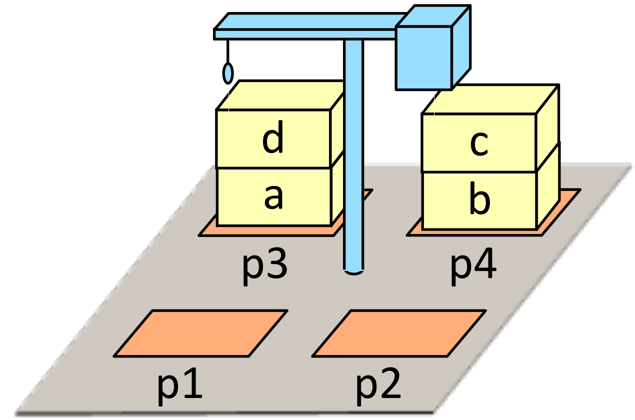
move(c, y, z)
  pre: pos(c)=y, clear(c)=T, clear(z)=T
  eff: pos(c)←z, clear(y)←T, clear(z)←F

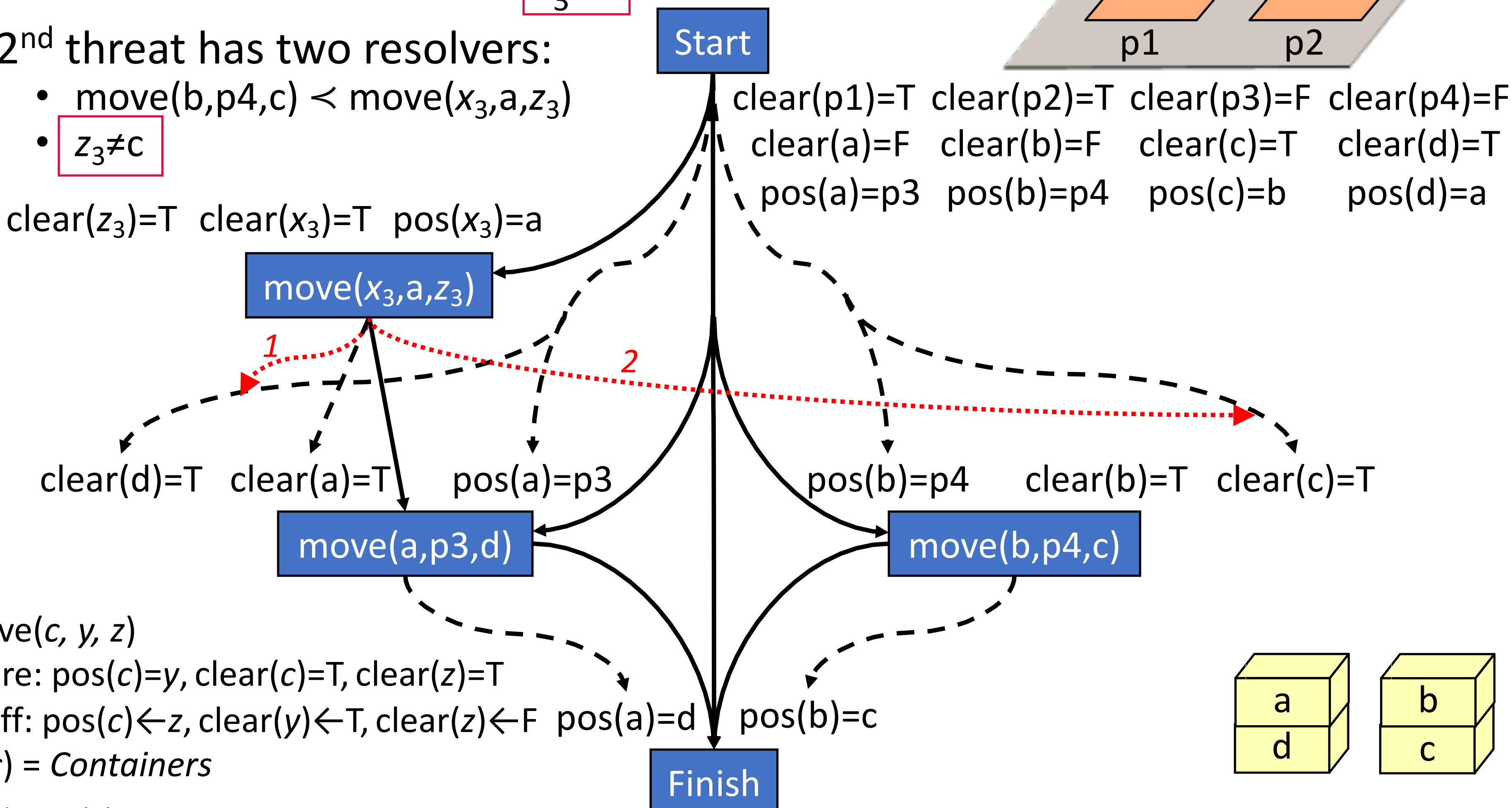
$\mathcal{R}(c)$ = *Containers*

$\mathcal{R}(y) = \mathcal{R}(z)$ = *Container* ∪ *pallets*

Example

- New action to resolve open goal
- 1^{st} threat has one resolver: $\boxed{z_3 \neq d}$
- 2^{nd} threat has two resolvers:
  - move(b,p4,c) $\prec$ move(x_3,a,z_3)
  - $\boxed{z_3 \neq c}$





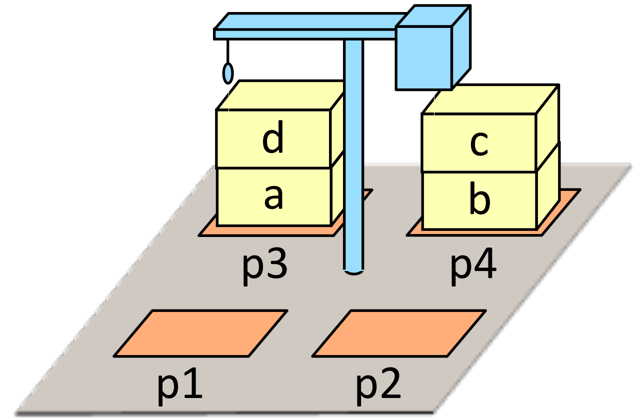
move(c, y, z)
pre: pos(c)=y, clear(c)=T, clear(z)=T
eff: pos(c)$\leftarrow z$, clear(y)$\leftarrow$T, clear(z)$\leftarrow$F

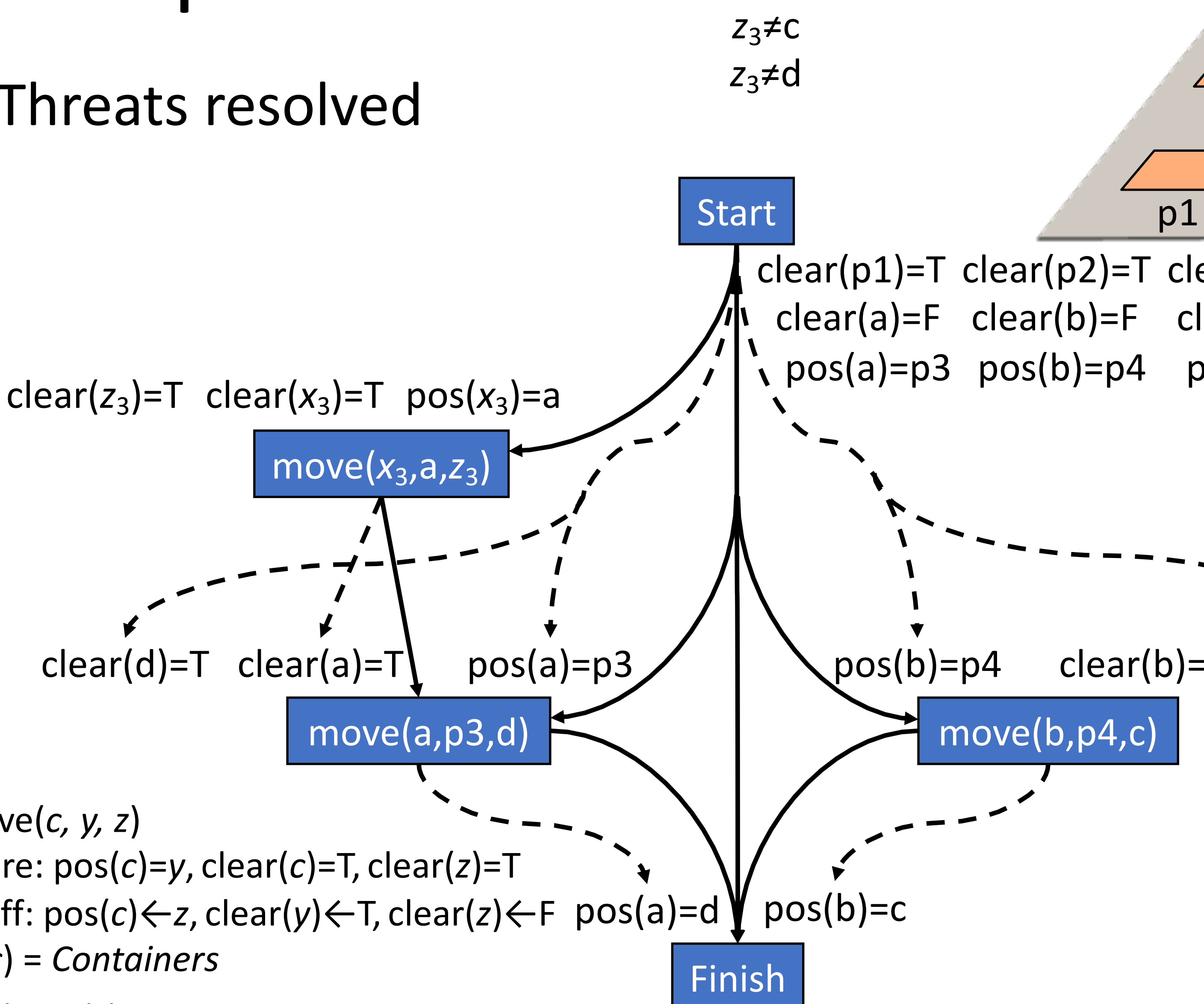
$\mathcal{R}(c)$ = *Containers*

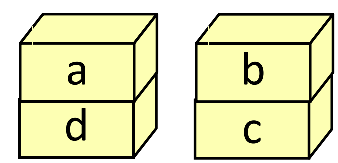
$\mathcal{R}(y) = \mathcal{R}(z)$ = *Container* ∪ *pallets*

Example

- Threats resolved







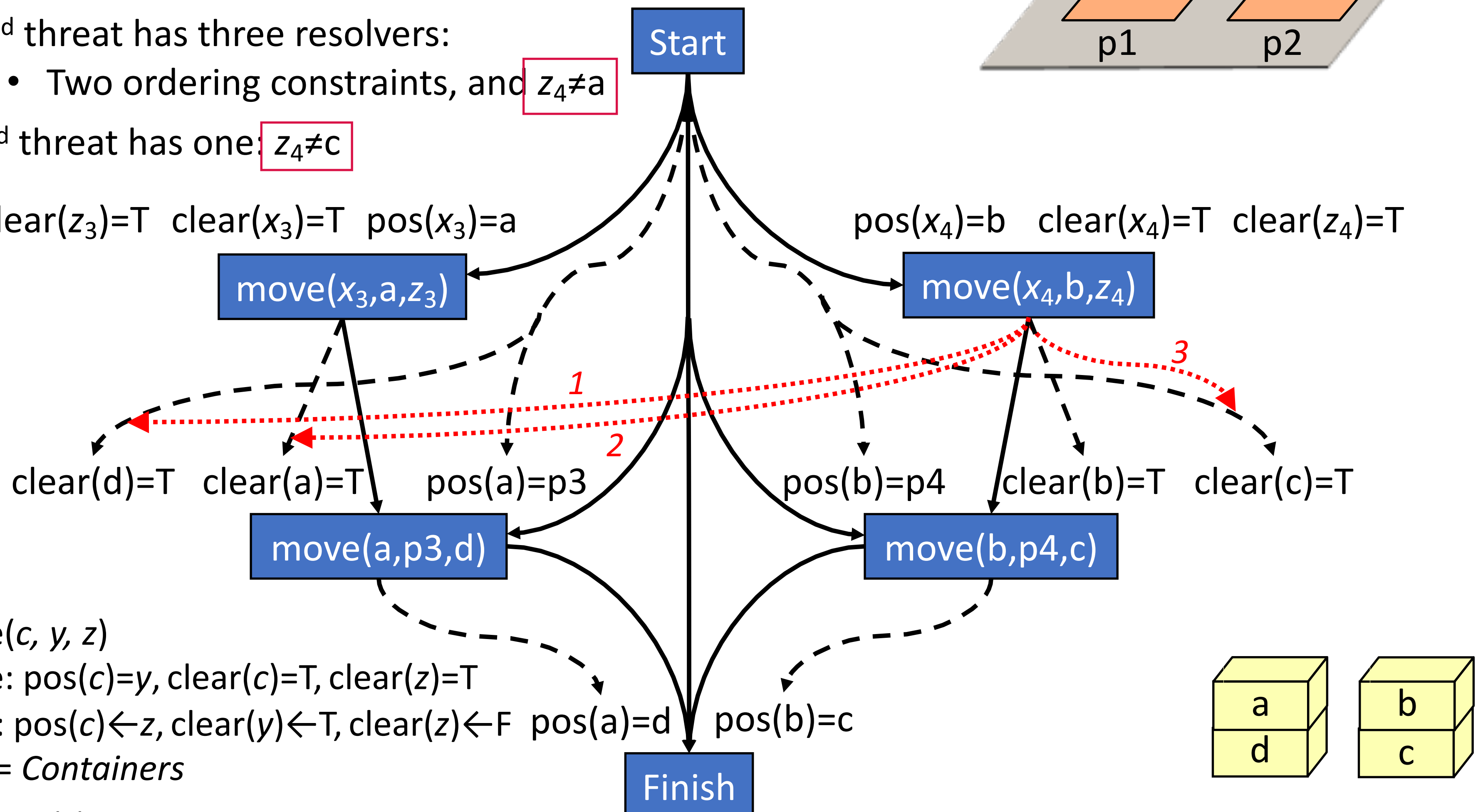
move(*c*, *y*, *z*)
pre: pos(*c*)=*y*, clear(*c*)=T, clear(*z*)=T
eff: pos(*c*)←*z*, clear(*y*)←T, clear(*z*)←F

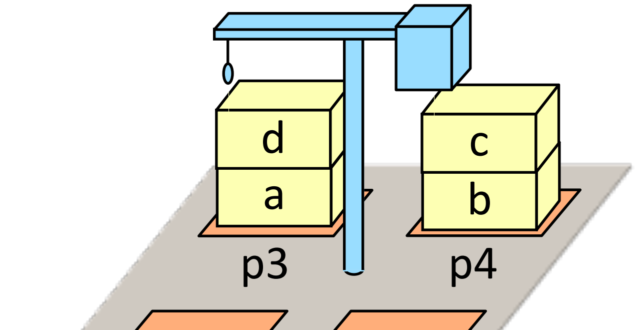
$\mathcal{R}(c)$ = *Containers*

$\mathcal{R}(y) = \mathcal{R}(z)$ = *Container* ∪ *pallets*

Example

- 1^{st} threat has two resolvers:
  - An ordering constraint, and $\boxed{z_4 \neq d}$
- 2^{nd} threat has three resolvers:
  - Two ordering constraints, and $\boxed{z_4 \neq a}$
- 3^{rd} threat has one: $\boxed{z_4 \neq c}$





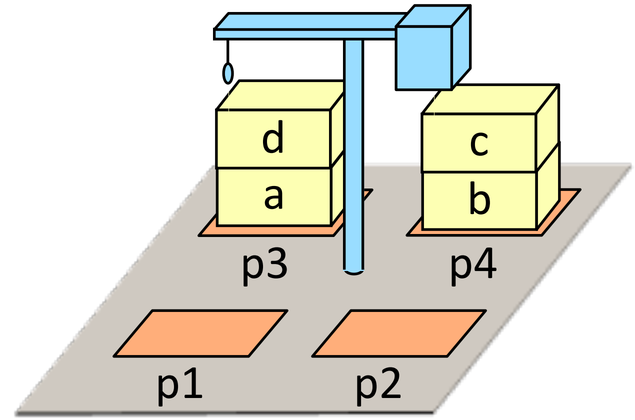
move(c, y, z)
  pre: pos(c)=y, clear(c)=T, clear(z)=T
  eff: pos(c)$\leftarrow z$, clear(y)$\leftarrow$T, clear(z)$\leftarrow$F

$\mathcal{R}(c)$ = *Containers*

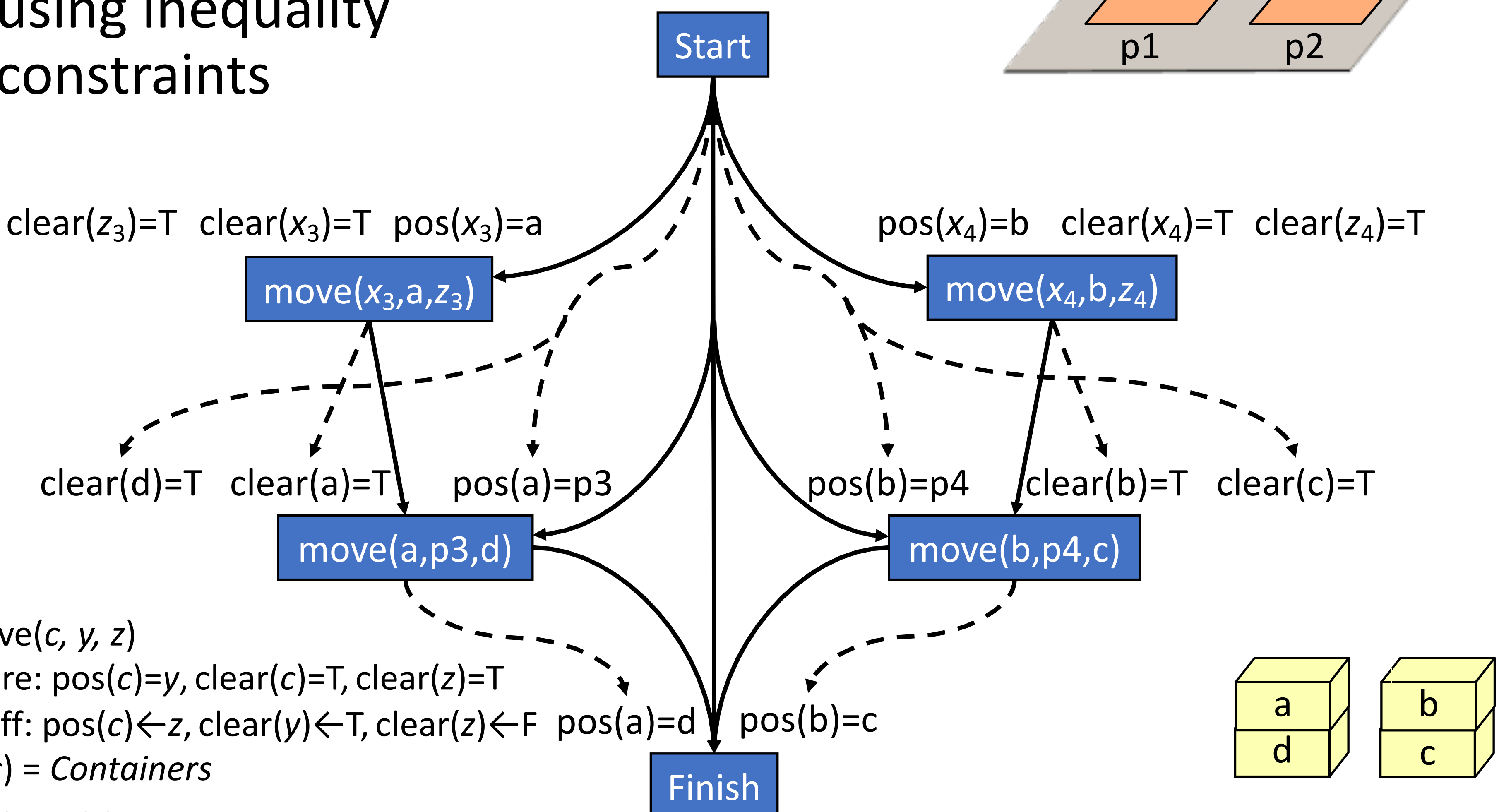
$\mathcal{R}(y) = \mathcal{R}(z)$ = *Container* $\cup$ *pallets*

Example

- Resolve the three threats using inequality constraints



$z_3 \neq c$ $z_4 \neq a$
$z_3 \neq d$ $z_4 \neq c$
$z_4 \neq d$

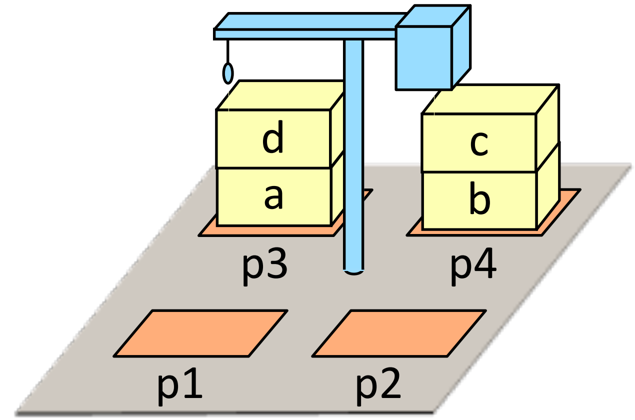


move(c, y, z)
  pre: pos(c)=y, clear(c)=T, clear(z)=T
  eff: pos(c)←z, clear(y)←T, clear(z)←F

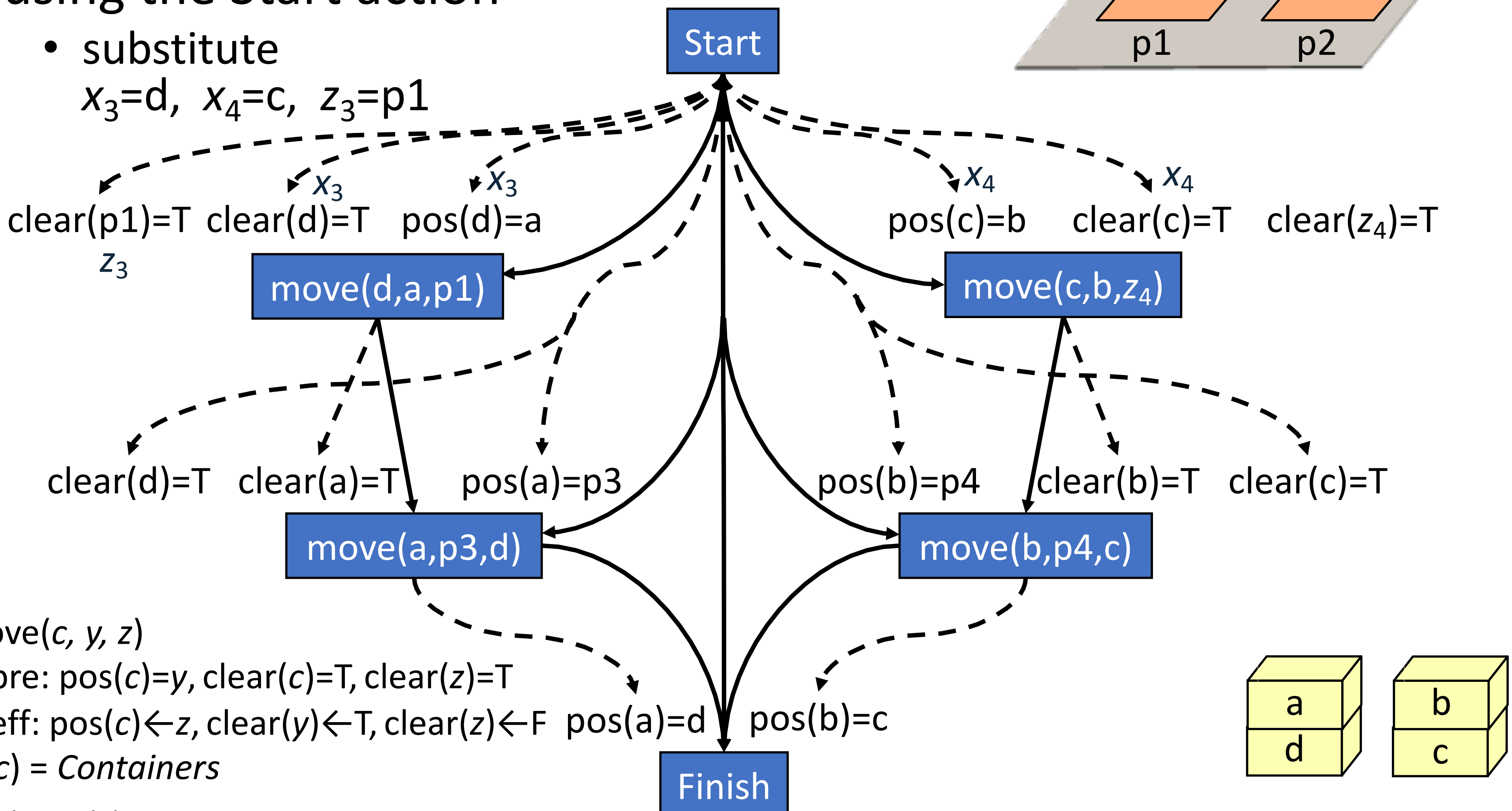
$\mathcal{R}(c)$ = *Containers*

$\mathcal{R}(y) = \mathcal{R}(z)$ = *Container* ∪ *pallets*

Example



- Resolve five open goals using the Start action
  - substitute
    x_3=d, x_4=c, z_3=p1



move(*c, y, z*)
  pre: pos(*c*)=*y*, clear(*c*)=T, clear(*z*)=T
  eff: pos(*c*)←*z*, clear(*y*)←T, clear(*z*)←F

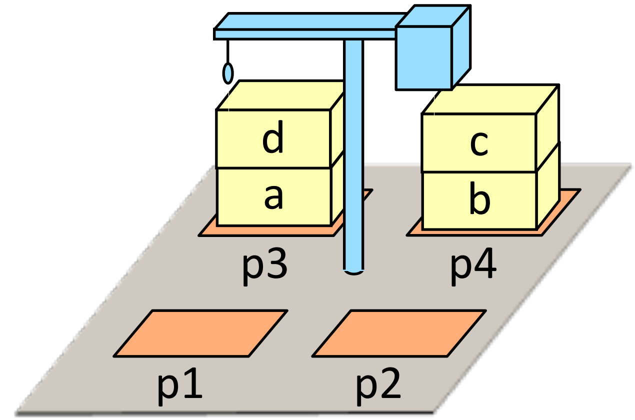
$\mathcal{R}(c)$ = *Containers*

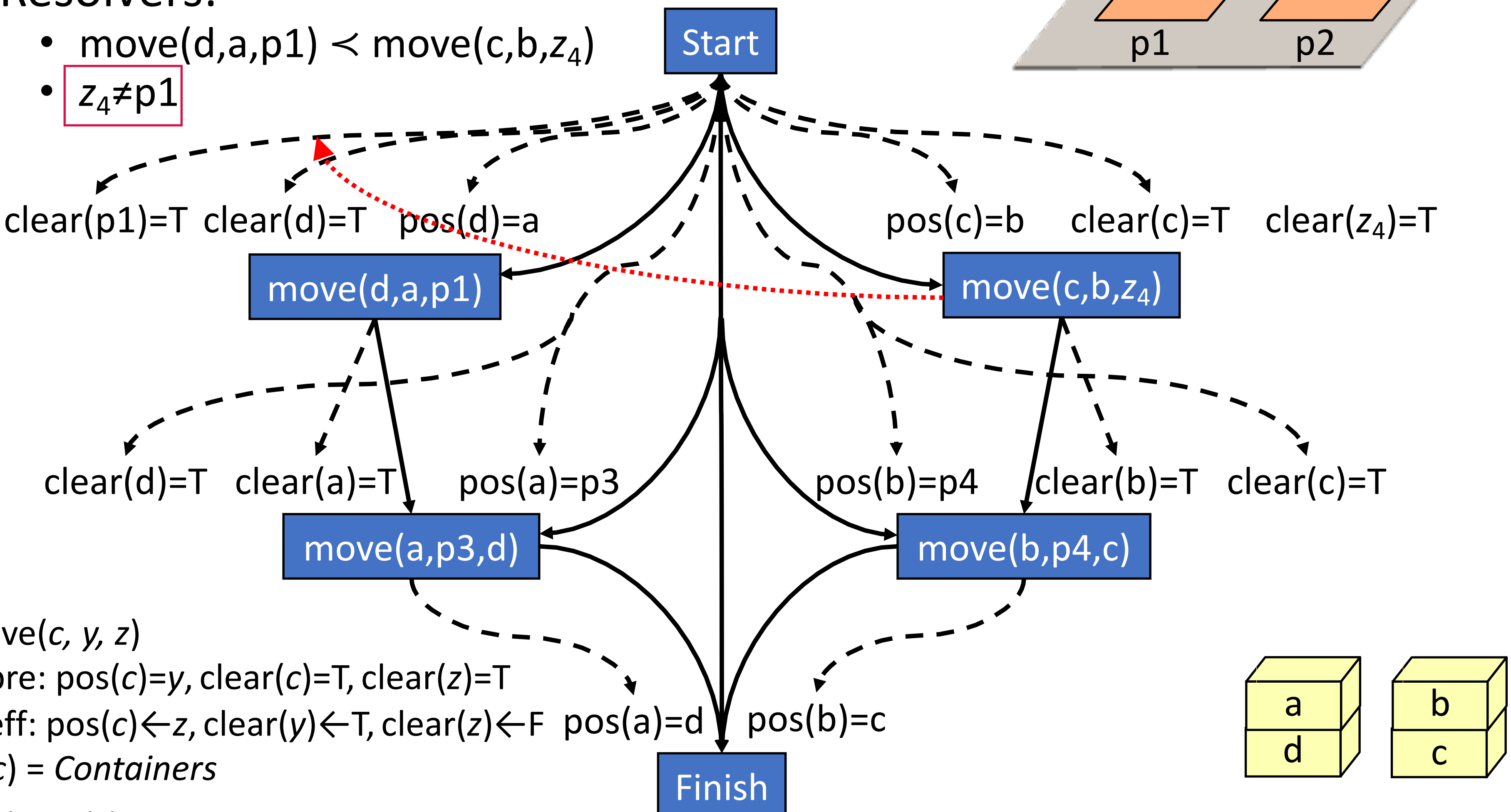
$\mathcal{R}(y) = \mathcal{R}(z)$ = *Container* ∪ *pallets*

Example

- Threatened causal link
- Resolvers:
  - move(d,a,p1) $\prec$ move(c,b,z_4)
  - $\boxed{z_4 \neq p1}$

$z_4 \neq a$
$z_4 \neq c$
$z_4 \neq d$





move(c, y, z)
pre: pos(c)=y, clear(c)=T, clear(z)=T
eff: pos(c)$\leftarrow z$, clear(y)$\leftarrow$T, clear(z)$\leftarrow$F

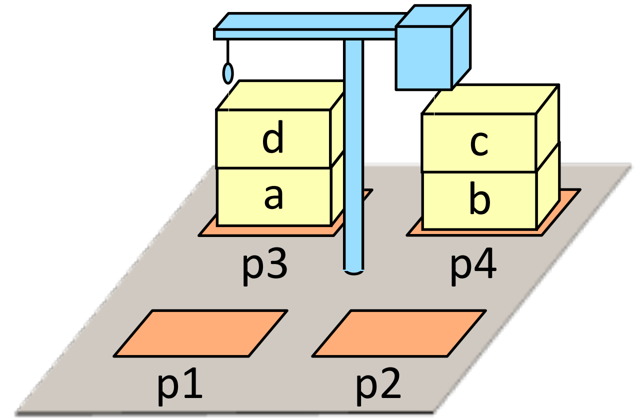
$\mathcal{R}(c)$ = *Containers*

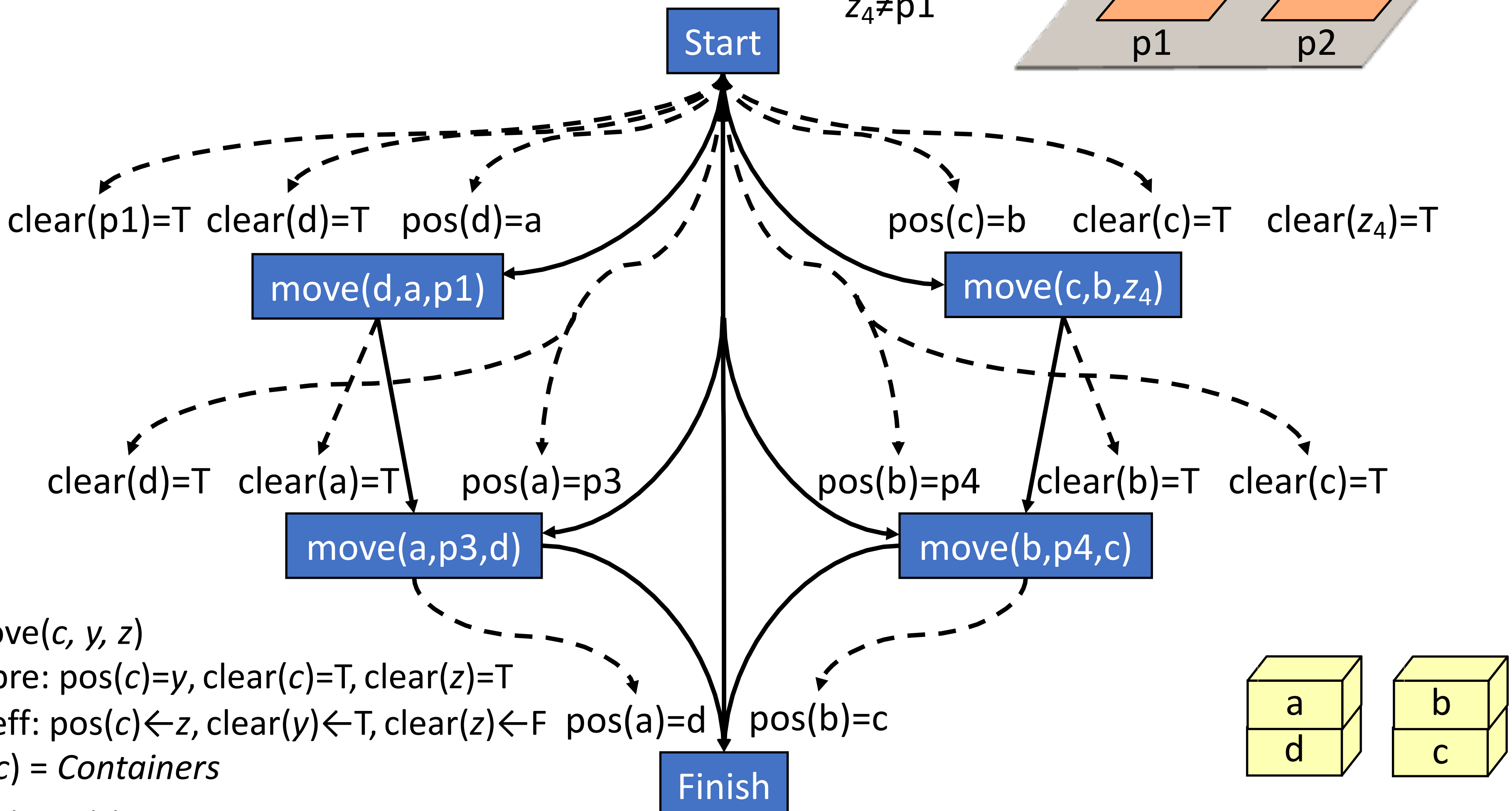
$\mathcal{R}(y) = \mathcal{R}(z)$ = *Container* $\cup$ *pallets*

Example

- Threat resolved

$z_4 \neq a$
$z_4 \neq c$
$z_4 \neq d$
$z_4 \neq p1$





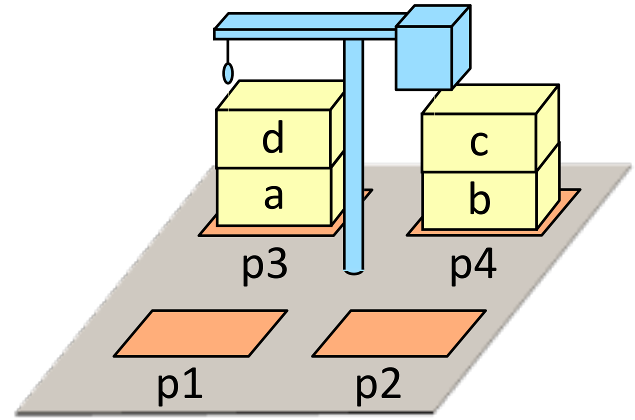
move(c, y, z)
pre: pos(c)=y, clear(c)=T, clear(z)=T
eff: pos(c)←z, clear(y)←T, clear(z)←F

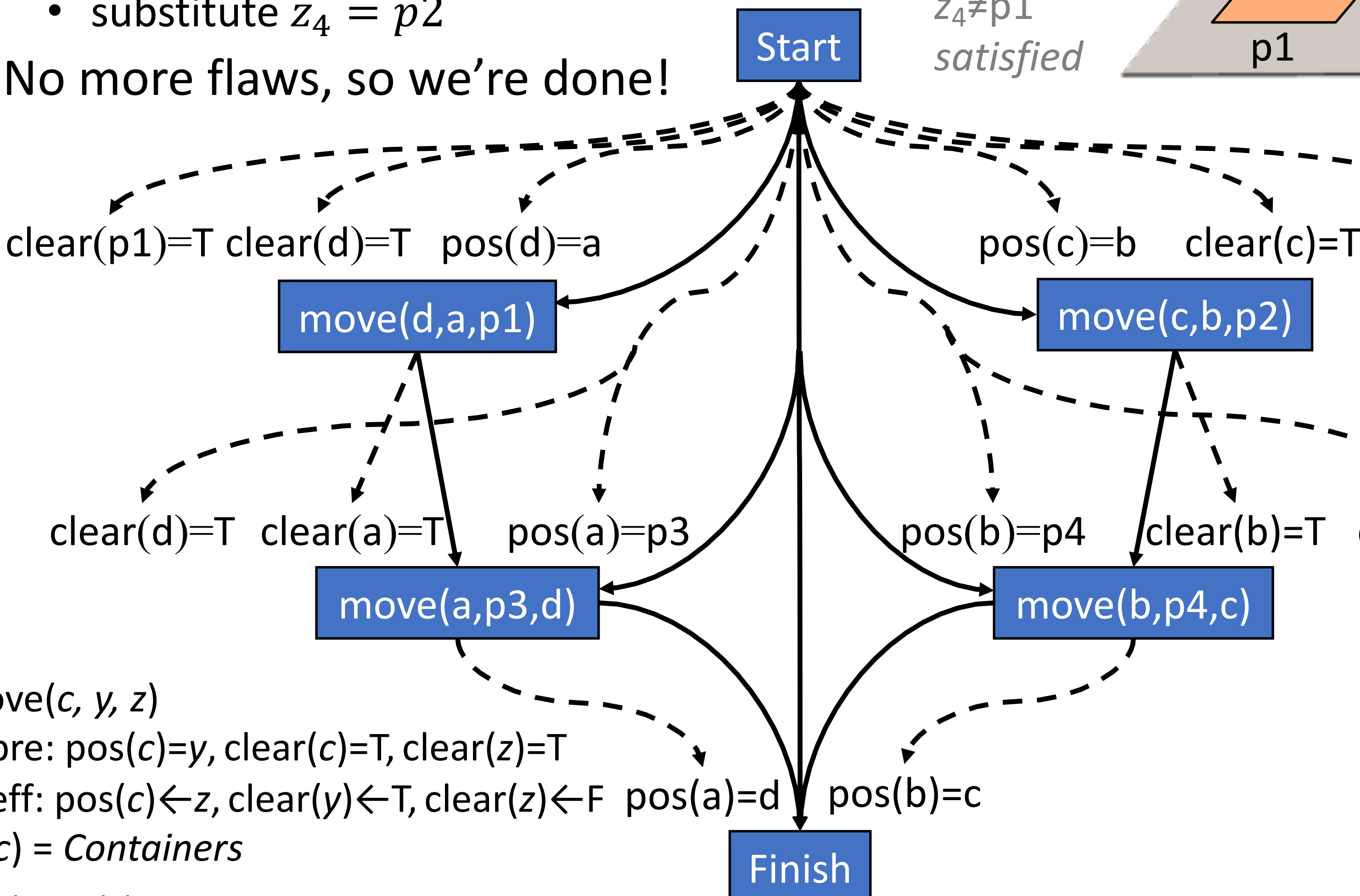
$\mathcal{R}(c)$ = *Containers*

$\mathcal{R}(y) = \mathcal{R}(z)$ = *Container* ∪ *pallets*

Example

- Resolve open goal using the Start action
  - substitute $z_4 = p2$
- No more flaws, so we're done!

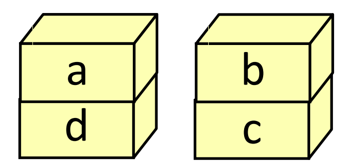




move(*c, y, z*)
  pre: pos(*c*)=*y*, clear(*c*)=T, clear(*z*)=T
  eff: pos(*c*)←*z*, clear(*y*)←T, clear(*z*)←F

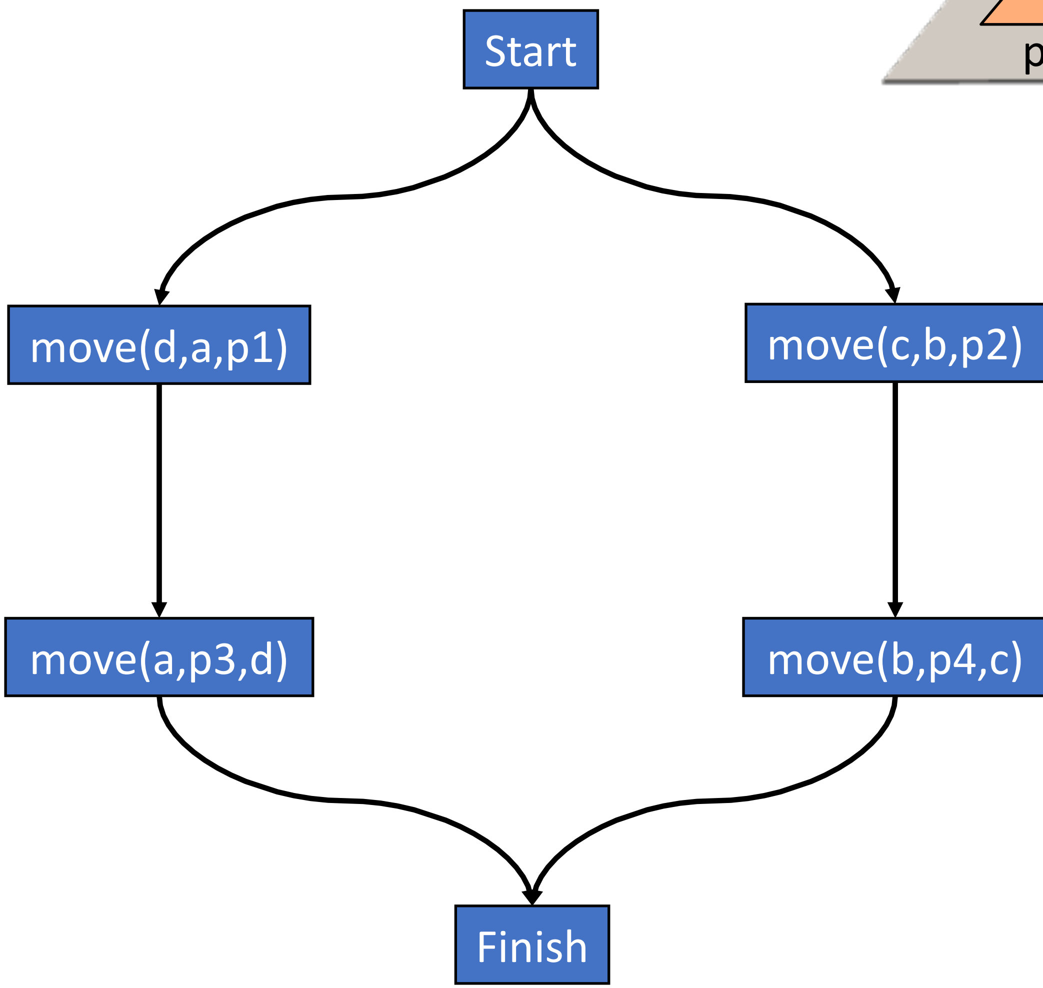
$\mathcal{R}(c)$ = *Containers*

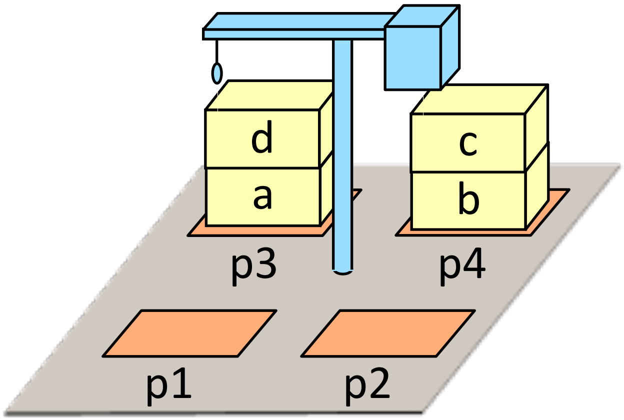
$\mathcal{R}(y) = \mathcal{R}(z)$ = *Container* ∪ *pallets*

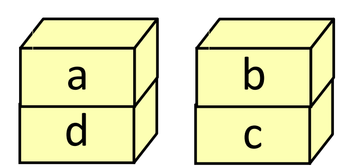


Example

- PSP returns this solution:

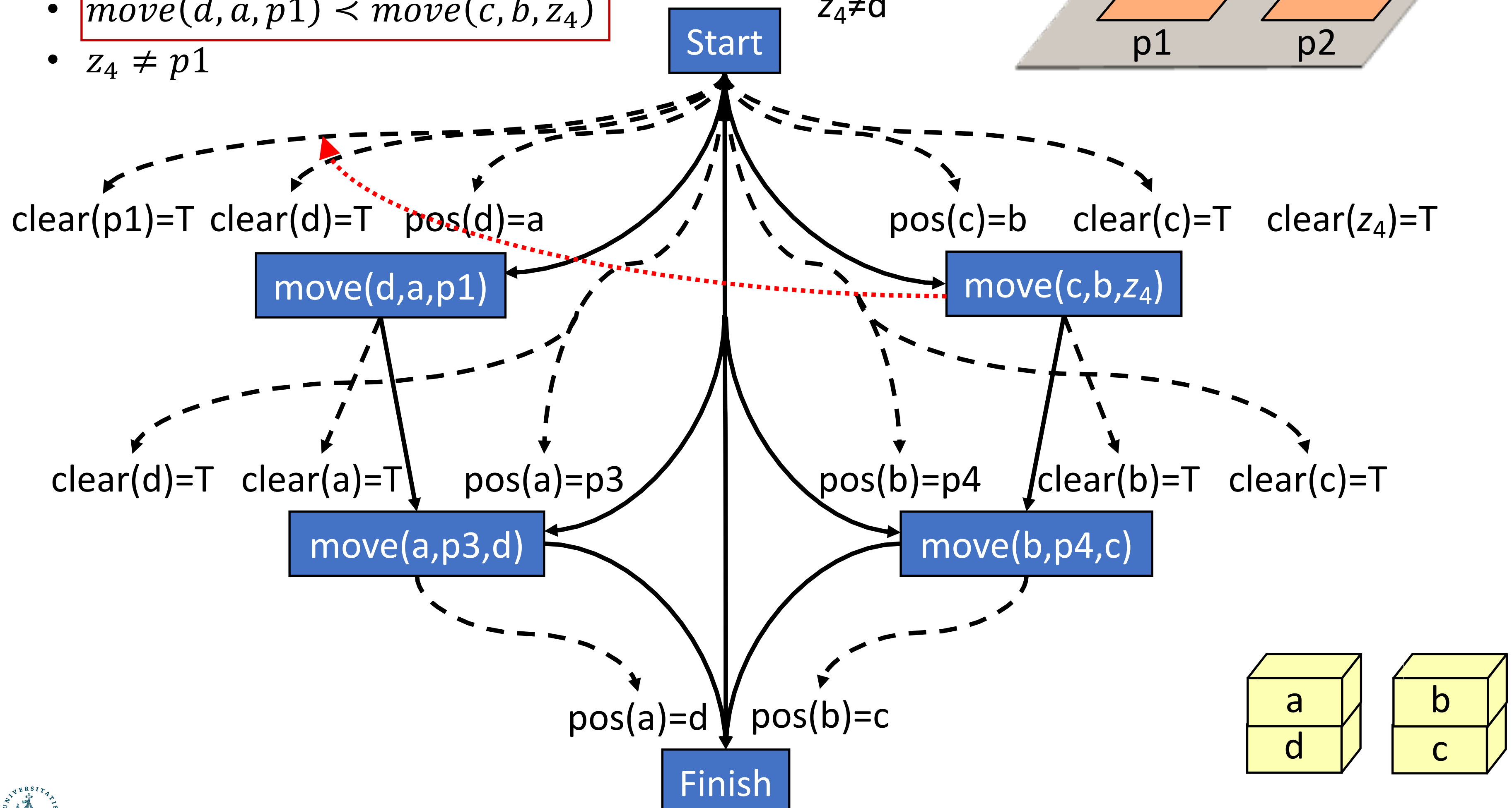


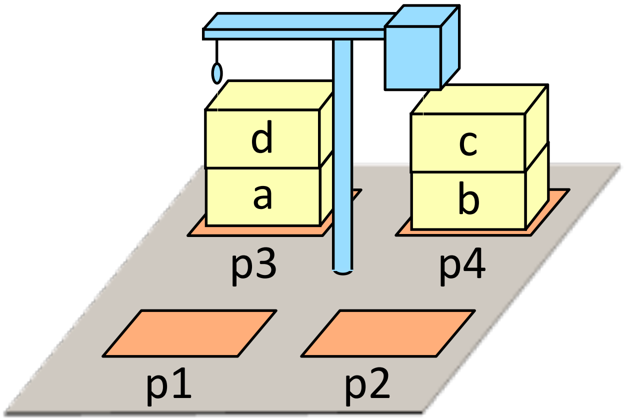




Example 2

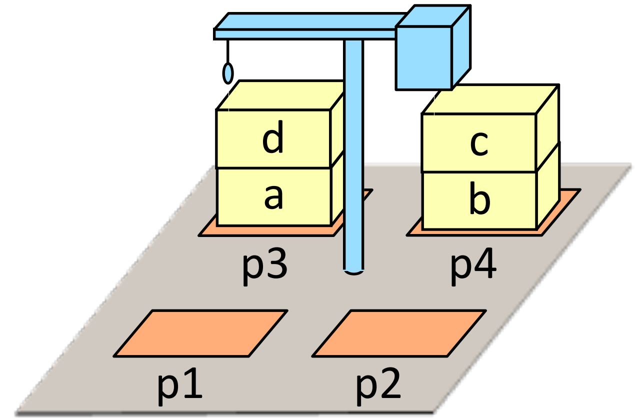
- Go back to the last threat, choose the other resolver:
  - $\boxed{move(d,a,p1) \prec move(c,b,z_4)}$
  - $z_4 \neq p1$

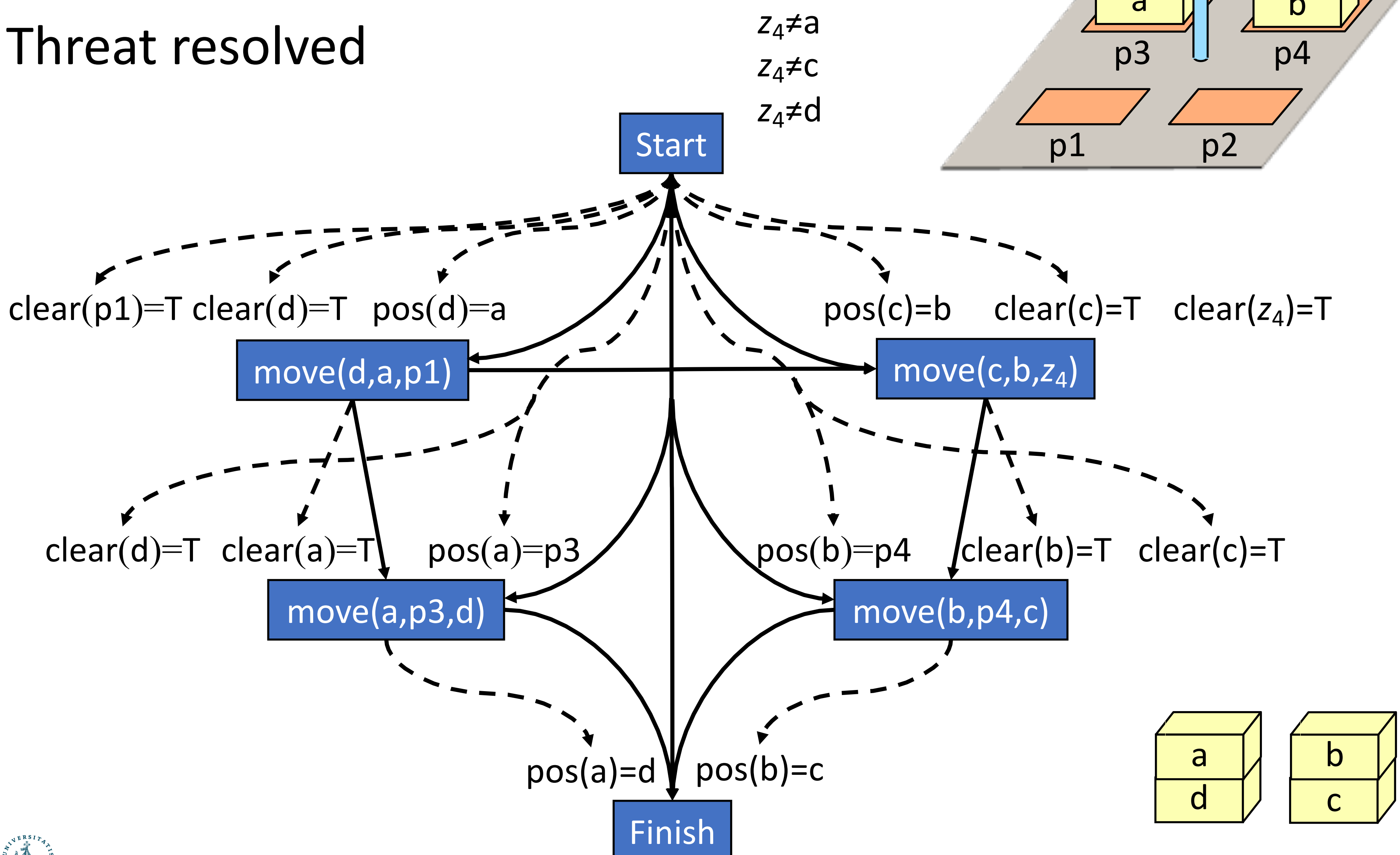




Example 2

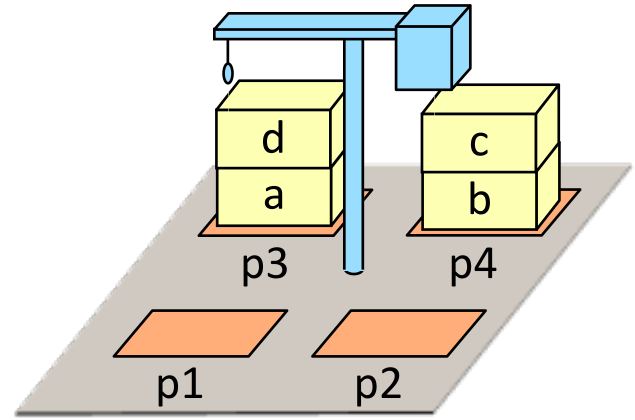
- Threat resolved



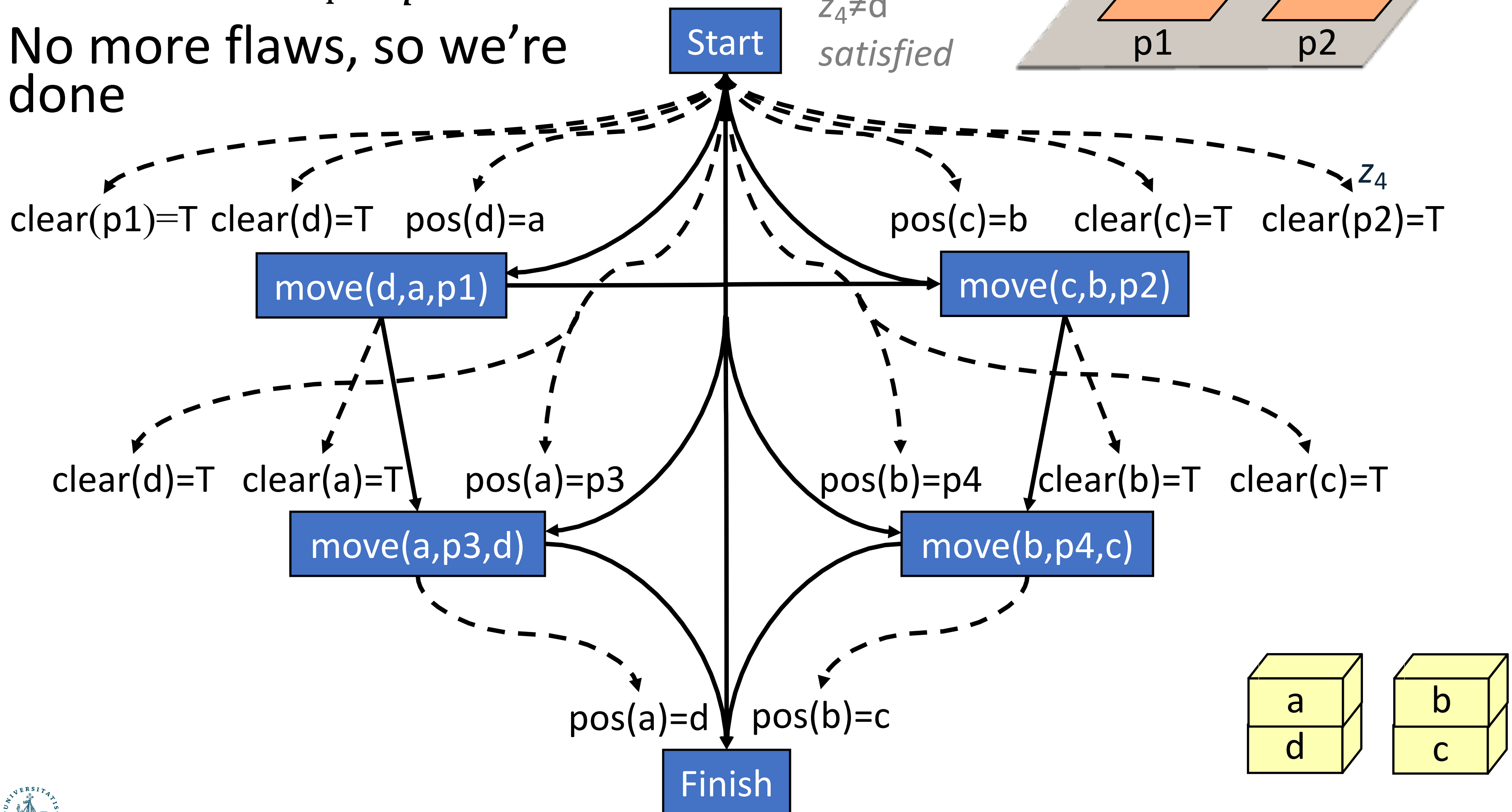


Example 2

- Resolve open goal
  - substitute $z_4 = p2$
- No more flaws, so we're done

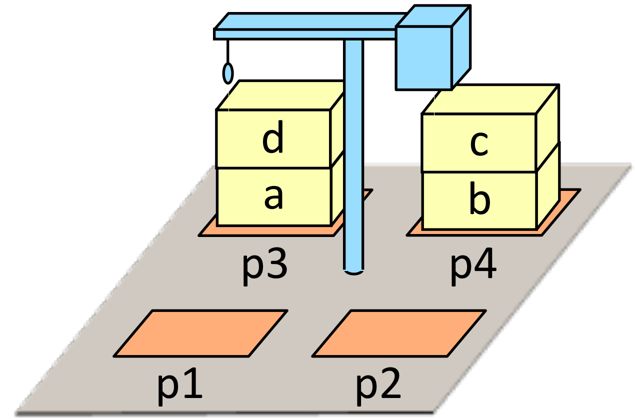


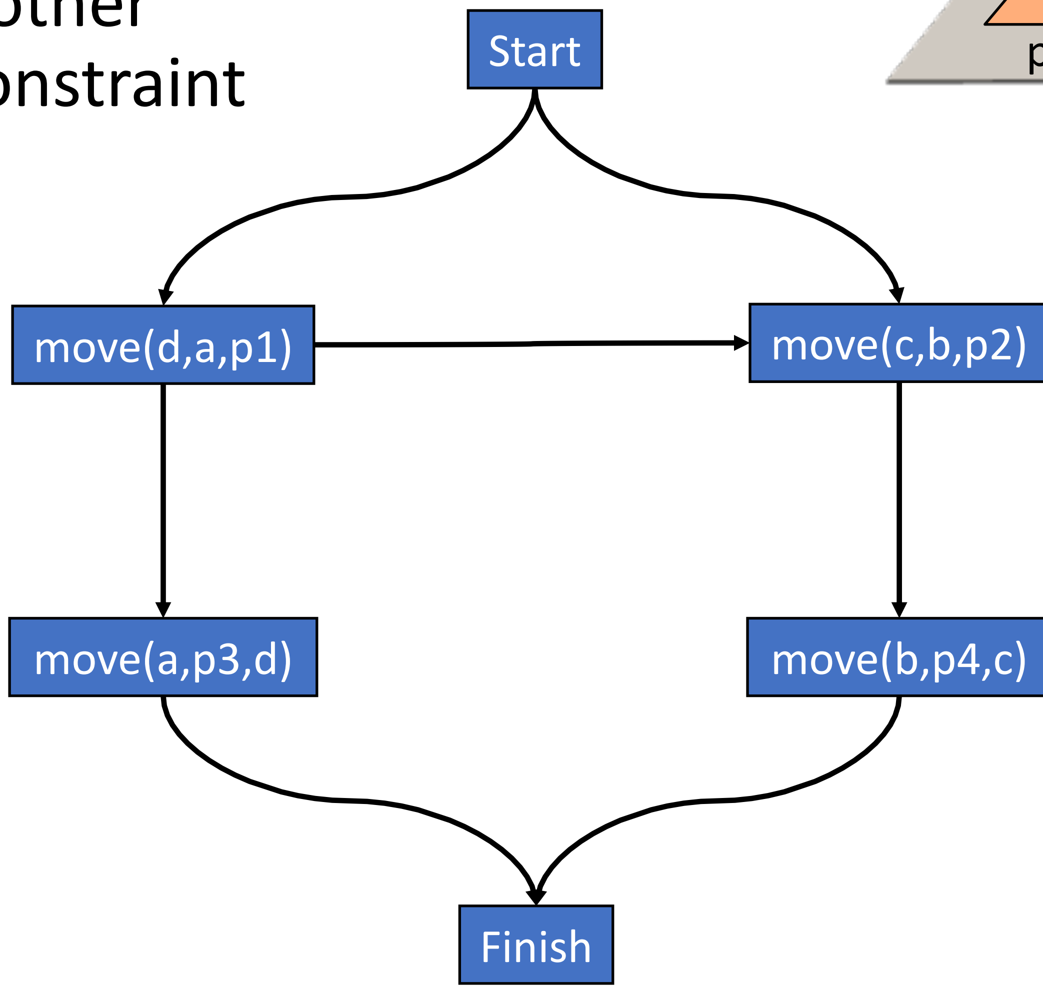
$z_4 \neq a$
$z_4 \neq c$
$z_4 \neq d$
satisfied

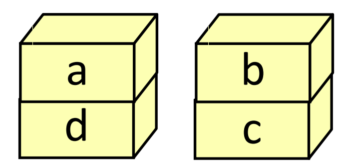


Example 2

- Like previous solution, but has another ordering constraint





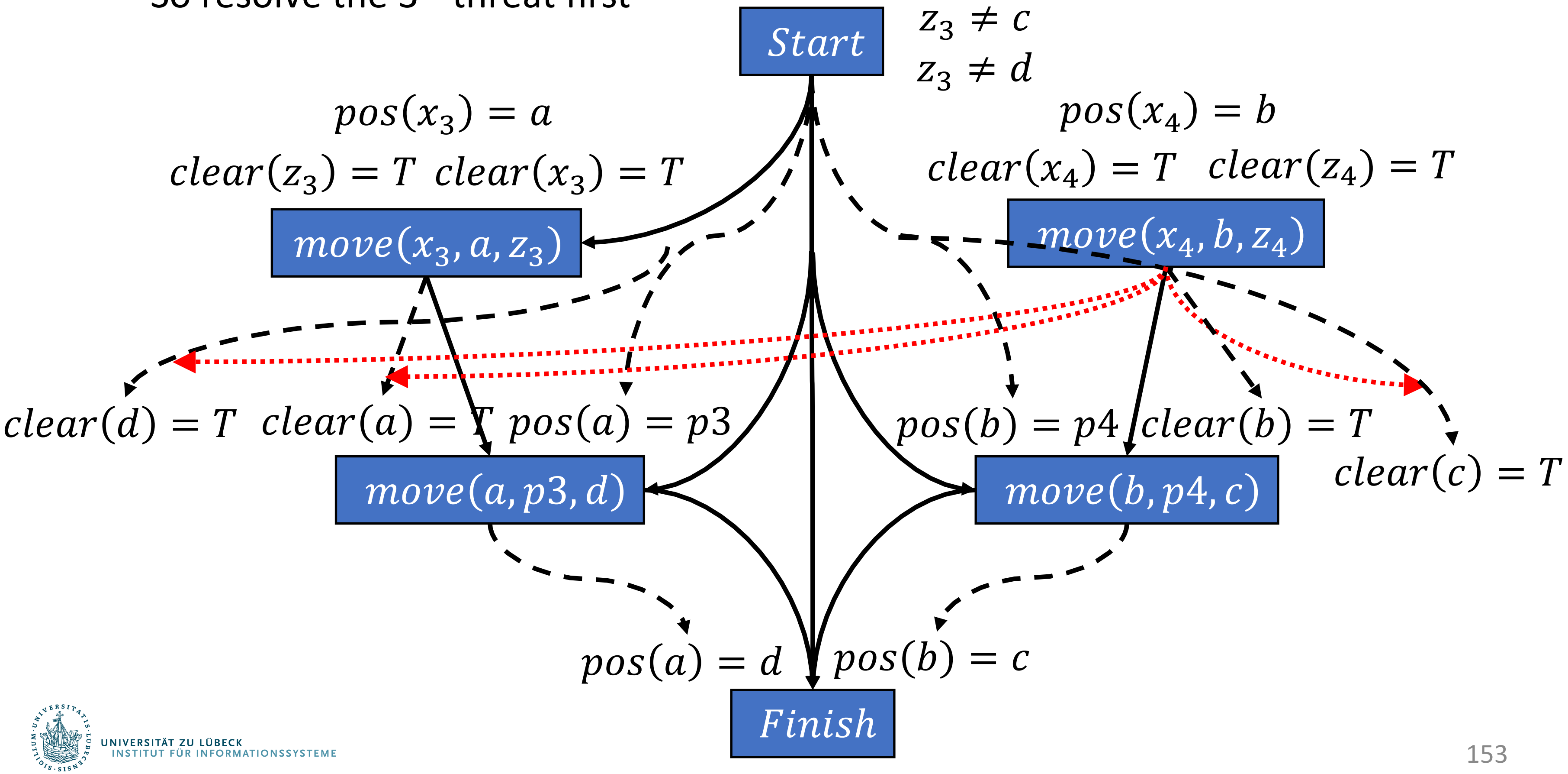


Node-selection Heuristics

- Analogy to constraint-satisfaction problems (CSPs)
  - Resolving a flaw in PSP
    ≈ assigning a value to a variable in a CSP
- What flaw to work on next?
  - Fewest Alternatives First (FAF)
    - Choose a flaw having the fewest resolvers
      ≈ Minimum Remaining Values (MRV) heuristic for CSPs
- To resolve the flaw, which resolver to try first?
  - Least Constraining Resolver (LCR)
    - Choose a resolver that rules out the fewest resolvers for the other flaws
      ≈ Least Constraining Value (LCV) heuristic for CSPs

Example

- Fewest Alternatives First:
  - 1st threat has two resolvers: an ordering constraint, and $z_4 \neq d$
  - 2nd threat has three resolvers: 2 ordering constraints, and $z_4 \neq a$
  - 3rd threat has one resolver: $z_4 \neq c$
- So resolve the 3rd threat first



Node-selection Heuristics

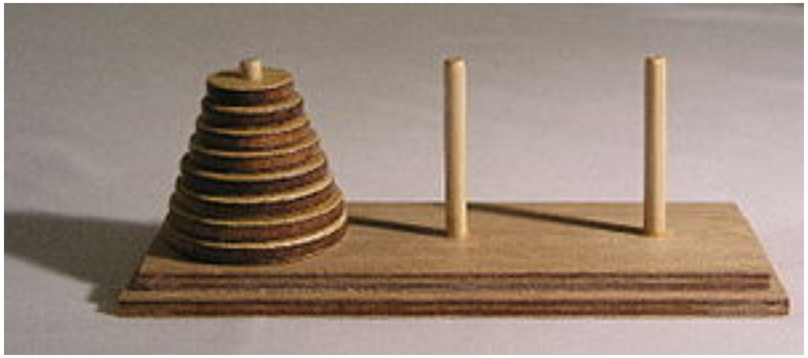
- In PSP, introducing a new action introduces new flaws to resolve
  - The plan can get arbitrarily large; want it to be as small as possible
    - Not like CSPs, where the search tree always has a fixed depth
  - Avoid introducing new actions unless necessary
- To choose between actions a and b, estimate distance from s_0 to $Pre(a)$ and $Pre(b)$
  - Can use the heuristic functions we discussed earlier

Discussion

- Problem: how to prune infinitely long paths in the search space?
  - Loop detection is based on recognizing states or goals we have seen before

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

  - In a partially ordered plan, we do not know the states
- Can we prune if π contains the same *action* more than once?
  - $\langle a1, a2, \ldots, a1, \ldots \rangle$
  - No. Sometimes we might need the same action several times in different states of the world
  - E.g., Towers of Hanoi problem
    - Do this action many times:
      - stack disk1 onto disk2



A Weak Pruning Technique

- Can prune all partial plans of n or more actions, where $n = |S|$
  - Not very helpful

"I'm not sure whether there's a good pruning technique for plan-space planning."

Dana Nau

Intermediate Summary

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

Summary

2.1 *State-variable representation*

- State = {values of variables}; action = changes to those values

2.2 *Forward state-space search*

- Start at initial state, look for sequence of actions that achieve goal

2.3 *Heuristic functions*

- How to guide a forward state-space search

2.6 *Incorporating planning into an actor*

- Online lookahead, unexpected events

2.4 *Backward search*

- Start at goal state, go backwards toward initial state

2.5 *Plan-space search*

- Start with incomplete plan for getting from initial state to goal state, make transformations to fix flaws in the plan

$\Longrightarrow$ Next: Planning and Acting with Refinement Methods