Lecture 5

CS 6260 Automated Planning and Learning

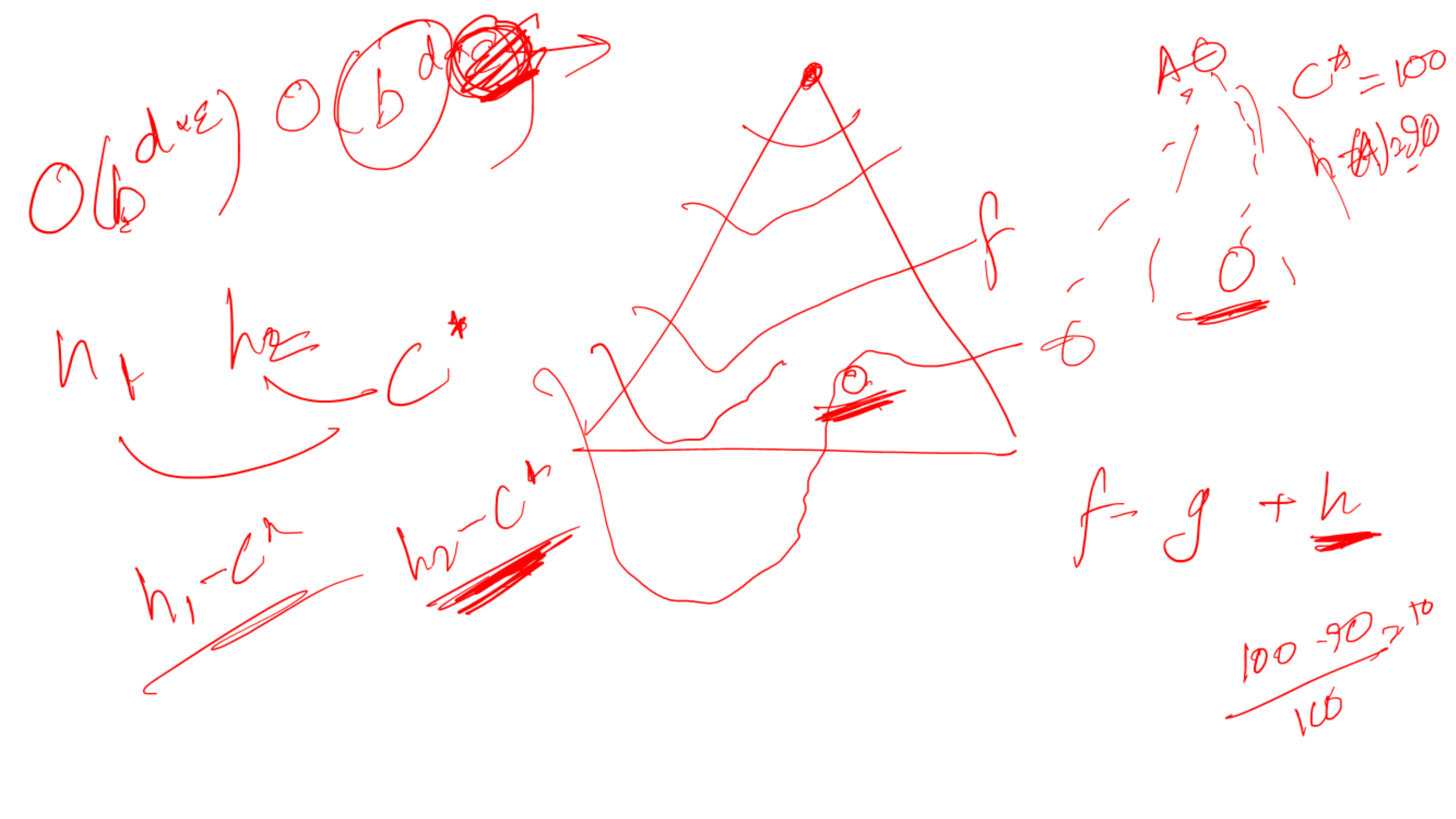
Classical Planning: Representation

Department of Computer Science and Engineering

Indian Institute of Technology Madras

Some material based on lecture slides for Automated Planning and Acting by Nau/Roberts.

Recap: A* Complexity


f
f - g + h

Classical Planning

State-Transition Systems in Planning

State-transition system or *classical planning domain*:
$\Sigma = (S,A,T,\text{cost})$ or (S,A,T)

- S - finite set of *states*
- A - finite set of *actions*
- $T: S \times A \to S$

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

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

  $T(s_0, a_1) = s_1$
  $T(s_1, a_2) = s_2$
  $\ldots$
  $T(s_{n-1}, a_n) = s_n$
  - In this case define $T(s_0, \pi) = s_n$, *and* $\hat{T}(s_0,\pi) = \langle s_1, \ldots, s_n \rangle$

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

Planning Problems

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

  $T(s_0, a_1) = s_1$

  $T(s_1, a_2) = s_2$

  …

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

  - In this case define $T(s_0, \pi) = s_n$, *and* $\hat{T}(s_0, \pi) = \langle s_1, \ldots, s_n \rangle$

- *Classical planning problem*:
  - $P = (\Sigma, s_0, S_g)$
  - planning domain, initial state, set of goal states
- *Solution* for *P*:
  - a plan π such that that $T(s_0, \pi) \in S_g$
- *Solution* for *P*: a plan π such that that $T(s_0, \pi) \in S_g$
  - *Minimal solution*: no subsequence is also a solution
  - *Shortest solution*: no solution has fewer actions
  - *Optimal solution*: no solution has lower cost
- **Example:** Suppose *P* has three solutions
  - $\pi_1 = \langle a_1 \rangle$
  - $\pi_2 = \langle a_2, a_3, a_4, a_5 \rangle$
  - $\pi_3 = \langle a_2, a_3, a_1 \rangle$
- Then π_1 is both shortest and optimal

$T(s_0, a_2) = s_{100}$

$T(s_{100}, a_3) = s_{92}$

$s \in S_g$

Which solutions are minimal?

A. π_1 B. π_2 C. π_3 D. A & B E. B & C

Acting with a Plan

- Run-Plan(Σ, π):

```
    while True do
1       s ← observe current state
        if π = ⟨⟩ then
2           return success
        a ← pop(π)
3       if a ∉ Applicable(s) then return failure
        perform action a
```

- *Where π is a solution to the classical planning problem:*
  - $P = (\Sigma, s_0, S_g)$
  - planning domain Σ, initial state s_0, set of goal states S_g
  - Recall: $\Sigma = (S,A,T,C)$ or (S,A,T)

Acting with a Plan

- Run-Plan(Σ, π):

```
    while True do
1       s ← observe current state
        if π = ⟨⟩ then
2           return success
        a ← pop(π)
3       if a ∉ Applicable(s) then return failure
        perform action a
```

- *Where π is a solution to the classical planning problem:*
  - $P = (\Sigma, s_0, S_g)$
  - planning domain Σ, initial state s_0, set of goal states S_g
  - Recall: $\Sigma = (S,A,T,C)$ or (S,A,T)

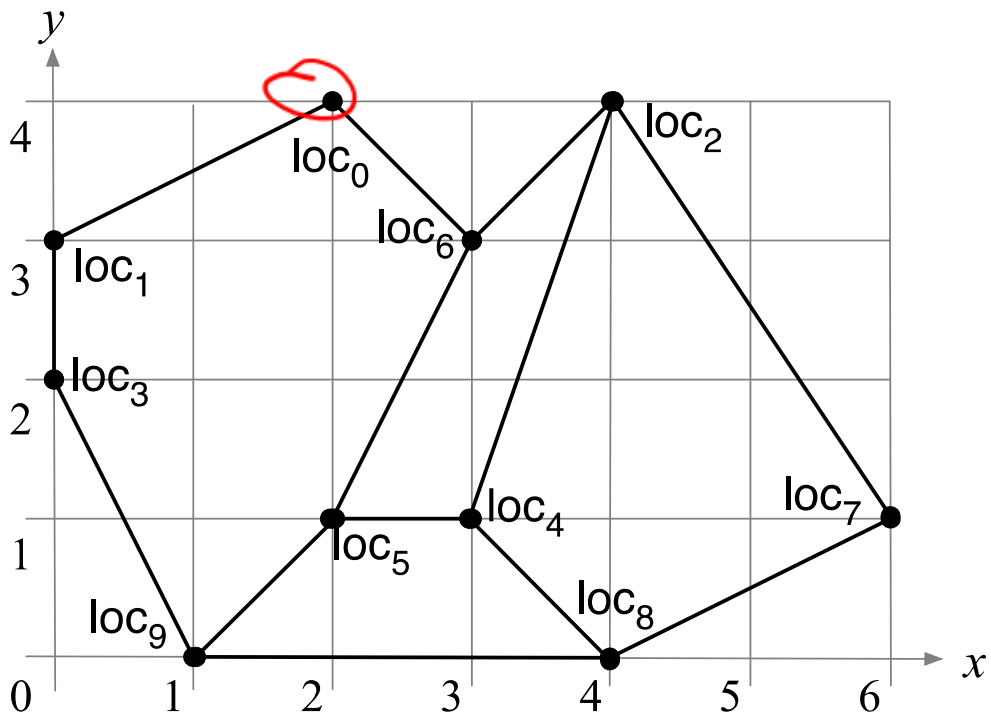
- Ideally, Run-Plan(Σ, $\langle a_1, \ldots, a_n \rangle$) will take Σ through through the sequence of states
  - $\hat{T}(s_0,\pi) = \langle s_1,\ldots, s_n \rangle$

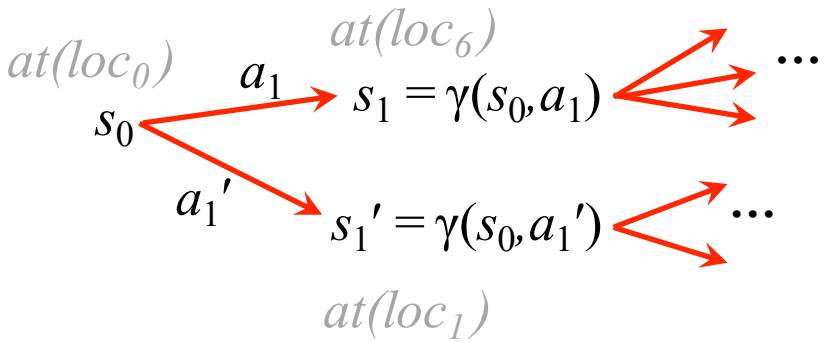
  then return success
- Note that Σ is unlikely to be a perfect model of the actor's environment
  - Later we'll discuss some things that can go wrong
- To test whether π has achieved a desired goal S_g
  - add S_g as a third argument
  - before line 2, insert this:

    if $s \notin S_g$ **then return** failure

Representation

- We write Run-Plan(Σ, π)
  - But what Run-Plan really needs is data structures that represent Σ and π
- If *S* and *A* are small enough
  - Give each state and action a name
  - For each *s* and *a*, store T(*s*,*a*) in a lookup table
- In larger domains, don't represent all states explicitly, so we need:
  - A language for describing properties of states
  - A language for describing how each action changes those properties
  - A procedure: start with an initial state, use actions to produce other states





Kinds of Representations

- *Domain-specific* representation:
  - tailor-made for a specific environment
- State: arbitrary data structure
- Action: (header, preconditions, effects, cost)
  - *header*: name and parameter list
    - Get actions by instantiating the parameters
  - *preconditions*:
    - Computational tests to predict whether an action can be performed
    - Should be necessary/sufficient for the action to run without error
  - *effects*:
    - Procedures that modify the current state
  - *cost*: procedure that returns a number
    - Can be omitted, default is cost ≡ 1

- Advantage: can use whatever works best for that particular domain
- Disadvantage: for each new domain, need new representation, new algorithms

- Alternative: *domain-independent* representation
  - A "standard format" that can be used for many different planning domains
  - May limit representational capability, but will allow more general compute approaches
  - Domain-independent algorithms that work for anything in this format

- We'll use a *state-variable* representation …

Kinds of Representations

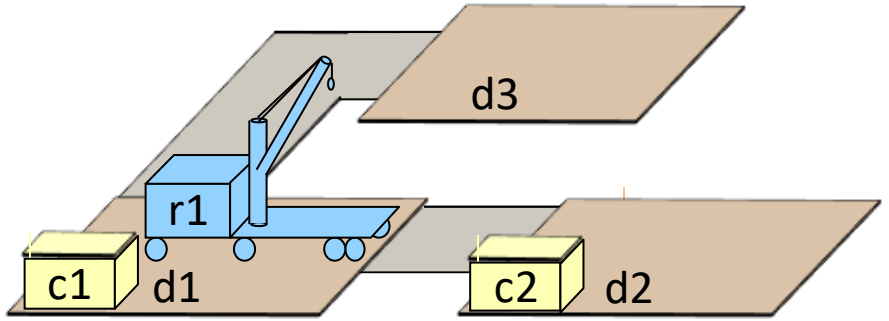
- Advantage of domain-specific representation:
  - use whatever works best for that particular domain
- Disadvantage:
  - for each new domain, need new representation and deliberation algorithms
- Alternative: *domain-independent* representation
  - Try to create a "standard format" that can be used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

State-Variable Representation

d_1 d_2 d_3 — docks

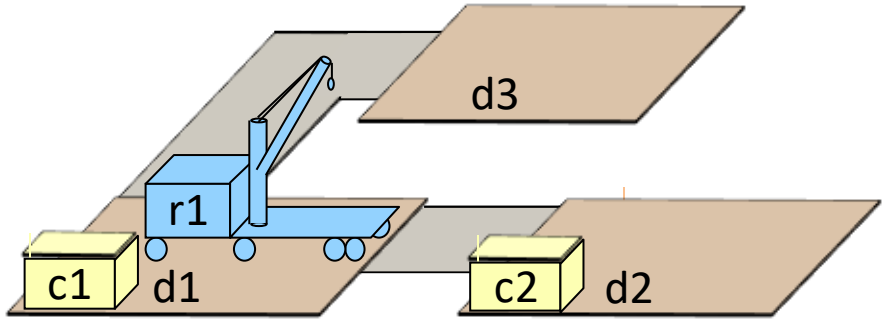
c_1 c_2 — containers

r_1 — robot



State-Variable Representation

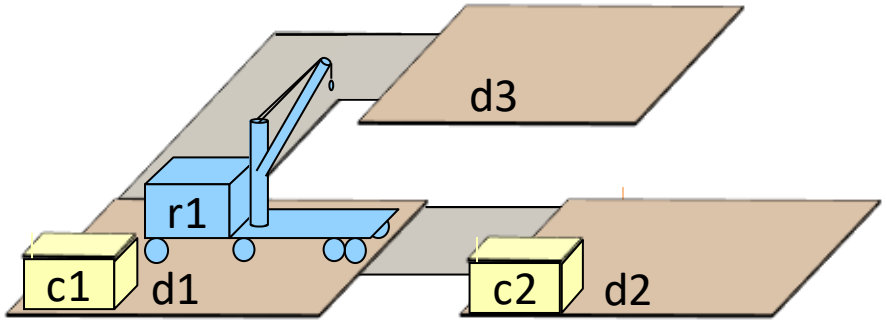
- *Objects* = {names of objects in the environment}
- Organized into a *typed ontology*
  - sets of object types
- *Objects* = *Robots* ∪ *Containers* ∪ *Locs* ∪ {nil}
  - *Robots* = {r1}
  - *Containers* = {c1, c2}
  - *Locs* = {d1, d2, d3}



- *Objects* only needs to include objects that matter at the current level of abstraction
- Can omit lots of details
  - physical characteristics of robots, containers, loading docks, roads, …
- Objects have two kinds of properties
  - *rigid* and *varying*

Rigid/Static Properties

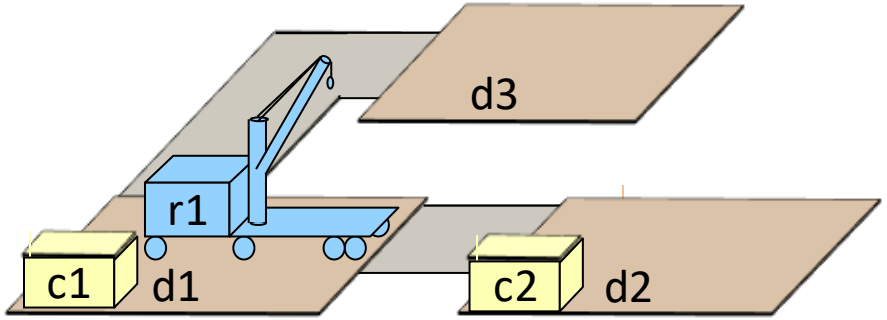
- *Rigid*: stays the same in every state
  - Can be described as a mathematical relation
    adjacent = {(d1,d2), (d2,d1), (d1,d3), (d3,d1)}
  - Or equivalently, a set of *ground atoms*
    adjacent(d1,d2), adjacent(d2,d1), adjacent(d1,d3), adjacent(d3,d1)
  - I'll use the two notations interchangeably



- Terminology from first-order logic:
  - *atom* ≡ *atomic formula* ≡ *positive literal* ≡ predicate symbol with list of arguments
    - e.g., adjacent(*x*,d2), where *x* is unbound
  - *negative literal* ≡ *negated atom* ≡ atom with a negation sign in front of it
    - e.g., ¬ adjacent(*x*,d2)
  - an atom is *ground* (or *fully instantiated*) if it contains no variable symbols, *lifted* if it contains all variable symbols, *partially instantiated* if it contains some variables
    - e.g., adjacent(d1,d2) is ground
    - e.g., adjacent(x,d2) is partially ground
    - e.g., adjacent(x,y) is lifted
  - *ground instance* of any expression: replace every variable with a value in its range
    - e.g., adjacent(d1,d2) is a ground instance of both adjacent(*x*,d2) and adjacent(*x*,*y*)

Varying Properties

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

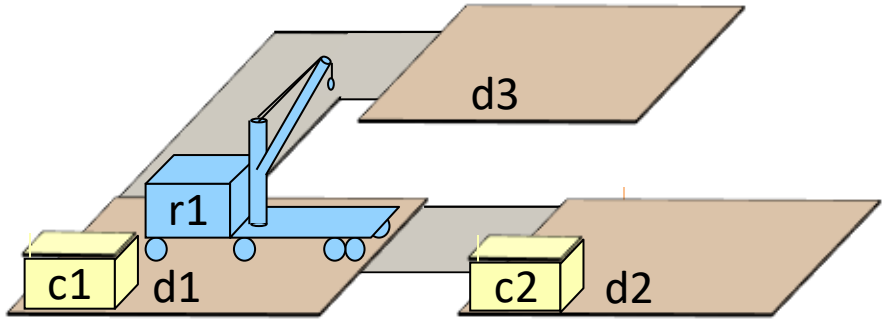


Instead of "domain", to avoid confusion with planning domains

States as Functions

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

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



- Mathematically, a function is a set of ordered pairs

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

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

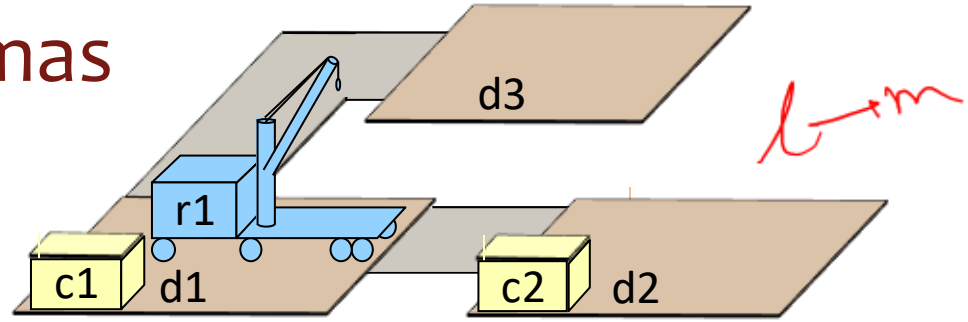
s_1 = {loc(r1)=d1, cargo(r1)=nil, loc(c1)=d1, loc(c2)=d2}

  - Here, '=' is a predicate symbol
  - Assignment will be indicated with '←'

Suppose r1 can move between locations. What does the state need to include to represent that move(r1, d1, d2) is valid while move(r1, d2, d3) is invalid?

(short answer)

Action Templates or Action Schemas



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

the *target* of the assignment

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

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

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

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

Planning Domain Definition Language [PDDL]

Will be covered in detail in Lecture 6

(:action move (?x ?y ?z)

pre: robot(?x) ∧
dock(?y) ∧ dock(?z)

adjacent(?x ?y)

and (robot (?x)
dock (?y)
dock (?z)
adjacent (?y ?z)
loc (?x ?y))

eff: ~~for~~
and (loc (?x ?z)
(not (loc (?x ?y)))

action move (?x - robot
?y - dock
?z - dock)

Will be covered in detail in Lecture 6

(: domain cargo-domain

(:predicates

~~adjacent (?x ?y)~~ adjacent (?x - dock ?y - dock)

loc (?x ?y)

cargo (?x)

robot (?x)

)

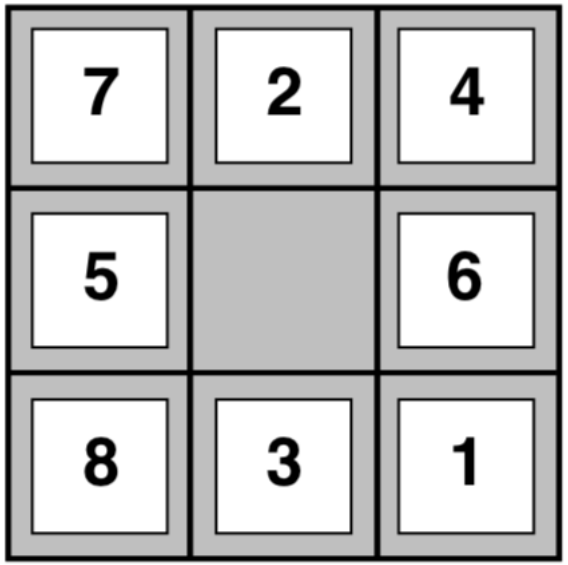
(:types cargo robot dock)

Will be covered in detail in Lecture 6

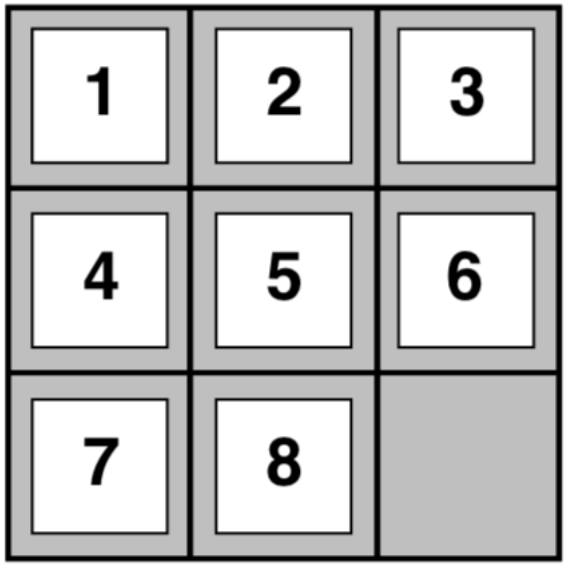
Generating Heuristics

Heuristic Estimates

- Easier in some problems than others!
  - Route finding problem has a natural metric (straight line distance)
  - Most state spaces are not so fortunate
- Online shopping agent
  - State space…
  - Metric?
- Household robot that enters a dark room
  - ”Distance” between a dark room and the same room with the light on

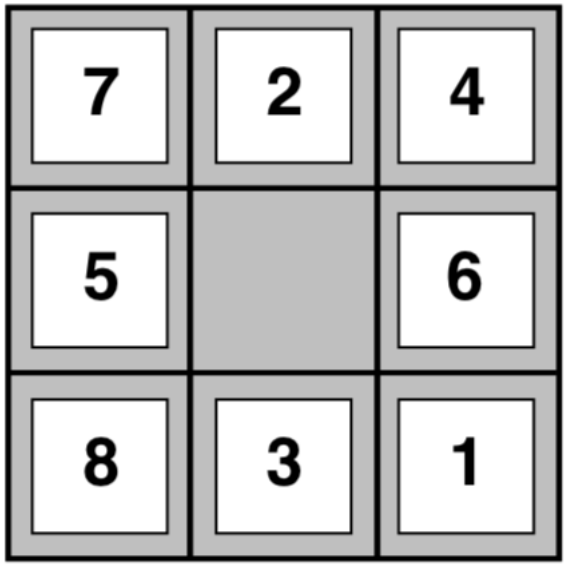


Start State

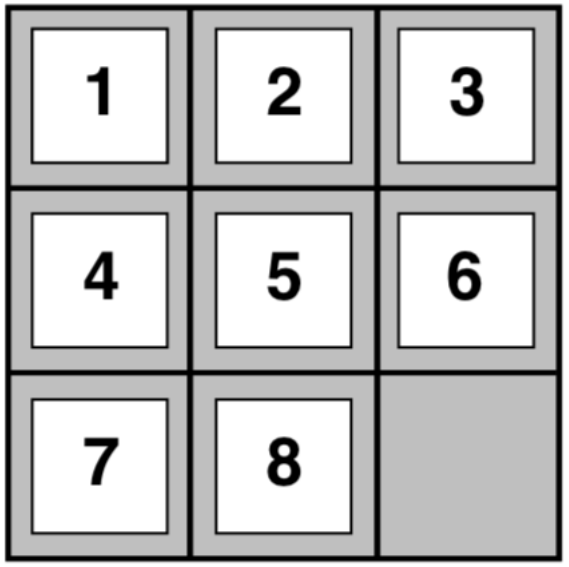


Goal State

- Can you think of an admissible heuristic?



Start State



Goal State

- Can you think of an admissible heuristic?
  - Number of misplaced tiles (6)
  - Total Manhattan distance ($4 + 0 + 3 + 3 + 1 + 0 + 2 + 1 = 14$)

Methodology for Generating Heuristics

- Create a relaxed version of the problem
  - Abstract, or approximate and easier
- Relaxed problems allow shortcuts
  - Optimal solution cost in relaxed $\leq$ optimal solution cost in real
- Optimal solution cost in the relaxed problem constitutes a heuristic
- But, if the relaxed problem is not easy, computing its optimal can be a challenge!

Where A* is Used in Practice…

- Video game route finding problems
- Robot path planning
- Natural Language Processing
- Speech recognition

Search Algorithms: Summary

- Use successor functions to formulate large search problems
- General notion of tree search and graph search
  - Fringe,
  - Node expansion strategies
- Informed vs uninformed search
  - If you can get a cost-to-goal estimate, use it as a heuristic
  - Admissible heuristics are good for guaranteed optimality
- Read further to learn about using inadmissible heuristics
  - (What would A* do?)
- To see some of these algorithms in action: https://movingai.com/SAS/