Automated Planning and Acting

Planning Domain & Problem Representation (Classical Planning)

Artificial Intelligence

olivier.boissier@emse.fr

INSTITUT Mines-Télécom

May 2019

MINES Saint-Étienne

Une école de l'IMT

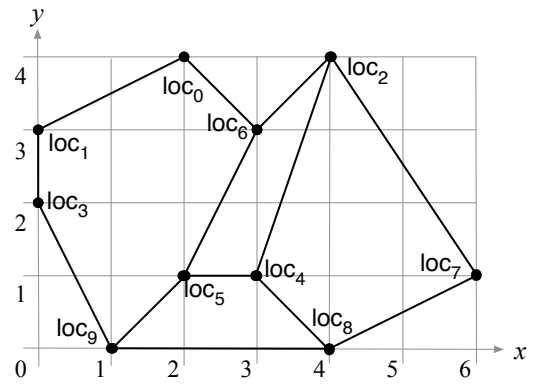
Outline

Domain Model

- *State-transition system* (or *classical planning domain*)
  - $\Sigma = (S,A,\gamma)$ or $\Sigma = (S,A,\gamma,\text{cost})$
    - S - finite set of *states* that the system may be in
    - A - finite set of *actions*: things the agent can do
    - γ: $S \times A \rightarrow S$ - *prediction function* (or *state-transition function*)
      - *partial* function: $\gamma(s,a)$ isn't defined unless a is *applicable* in s
      - $\text{Dom}(a) = \{s \in S \mid \gamma(s,a) \text{ is defined}\} = \{s \in S \mid a \text{ is applicable}\}$
      - $\text{Range}(a) = \{\gamma(s,a) \mid s \in \text{Dom}(a)\}$
    - cost: $S \times A \rightarrow \mathcal{R}$
      - could be monetary cost, time required, something else
- *Classical planning problem*: $P = (\Sigma, s_0, S_g)$
  - (Σ = planning domain, s_0 = initial state, S_g = set of goal states)
- *Solution* for P *is* a *plan* (i.e. sequence of actions) 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



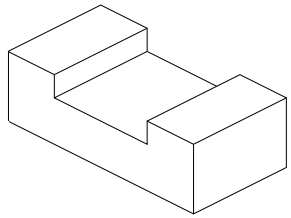
- In larger domains, don't represent all states explicitly
  - Language for describing properties of states
  - Language for describing how each action changes those properties
  - Start with initial state, use actions to produce other states

Domain-Specific Representation

- Design it specifically for the environment you need to model
- 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

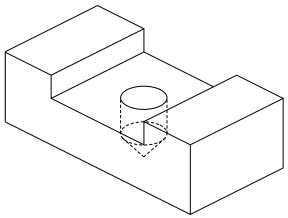
- Drilling holes in a metal workpiece
  - A state *s*
    - geometric model of the workpiece, info about its location and orientation
    - capabilities and status of drilling machine and drill bit
  - Several actions
    - getting the workpiece onto the machine
    - clamping it
    - loading a drill bit
    - etc.



- Next slide: the drilling operation itself

Drilling Operation

- Name and parameters: drill-hole($m, w, b, x_1, \ldots, x_n$)
  - dimensions, orientation, machining tolerances of the hole to be drilled)
- Preconditions
  - Can the drilling machine and drill bit produce a hole having the desired geometry and machining tolerances?
  - Is the drill loaded into the drilling machine? Is the workpiece is properly clamped onto the drilling platform? Etc.
- Effects
  - geometric model of how the workpiece geometry will be modified
- Cost
  - estimate of how much time will be needed, or how much money the action will cost



Discussion

☺ Advantage of domain-specific representation:
- can choose whatever works best for that particular domain

☹ Disadvantage:
- need new representation and deliberation techniques for each new domain

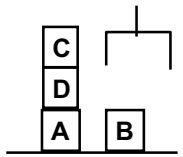
- Alternative: *domain-independent* representation
  - Try to create a “standard format” that can be used for many different planning domains
  - Deliberation algorithms that work for anything in this format
- *State-variable* representation
  - Simple formats for describing states and actions
  - Limited representational capability
    - But easy to compute, easy to reason about

Outline

Planning as Theorem Proving
Situation Calculus *(McCarthy, Hayes 1969)*

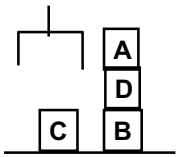
- Operators described by first-order logical formulas
  - Addition of an argument for the description of states
- Assumptions :
  - linear planning,
  - separation of planning and execution,
  - search in a space of formula,
  - no hypothesis on full knowledge of the initial, final state and operators,
  - Use of theorem proving to "prove" that a particular sequence of actions, when applied to a situation, leads to the expected result.

Situation Calculus - States



Initial state (Init)

on-table(A,Init)
on-table(B,Init)
on(D,A,Init)
on(C,D,Init)
clear(C,Init)
clear(B,Init)
arm-empty(Init)



Goal state (Goal)

on-table(B,Goal)
on-table(C,Goal)
on(D,B,Goal)
on(A,D,Goal)
clear(A,Goal)
clear(C,Goal)
arm-empty(Goal)

Situation Calculus - Operators

Descriptions of the changes in the world as a result of the agent's actions

PICKUP(x, ξ)
$\forall(x, \xi)$ clear(x, ξ) ∧ on-table (x, ξ) ∧ arm-empty(ξ) → holds(x, PICKUP(x, ξ)) ∧ ¬ arm-empty(PICKUP(x, ξ)) ∧ ¬ clear(x, PICKUP(x, ξ)) ∧ ¬ on-table(x, PICKUP(x, ξ))

PUTDOWN(x, ξ)
$\forall(x, \xi)$ holds(x, ξ) → clear(x, PUTDOWN(x, ξ)) ∧ on-table(x, PUTDOWN(x, ξ)) ∧ arm-empty(PUTDOWN(x,ξ)) ∧ ¬ holds(x, PUTDOWN(x, ξ))

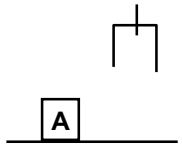
STACK(x,y, ξ)
$\forall(x,y, \xi)$ holds(x, ξ) ∧ clear(y, ξ) → clear(x, STACK(x,y, ξ)) ∧ on(x,y, STACK(x,y, ξ)) ∧ arm-empty(STACK(x,y,ξ)) ∧ ¬ holds(x, STACK(x,y, ξ)) ∧ ¬ clear(y, STACK(x,y, ξ))

UNSTACK(x,y, ξ)
$\forall(x,y, \xi)$ clear(x, ξ) ∧ on(x,y, ξ) ∧ arm-empty(ξ) → holds(x, UNSTACK(x,y, ξ)) ∧ clear(y, UNSTACK(x,y, ξ)) ∧ ¬ clear(x, UNSTACK(x,y, ξ)) ∧ ¬ on(x,y, UNSTACK(x,y, ξ)) ∧ ¬ arm-empty(UNSTACK(x,y, ξ))

Questions

- What is the resulting plan?

Situation Calculus - Example



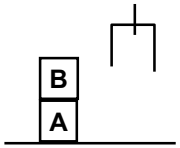
on-table(A,Init)
clear(A,Init)
arm-empty(Init)

holds(A,Goal)

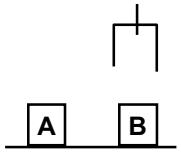
By applying PICKUP(A,Init) on the initial state:

holds(A,PICKUP(A,Init))
¬ arm-empty(A, PICKUP(A, Init))
¬ clear(A, PICKUP(A, Init))
¬ on-table(A, PICKUP(A, Init)) Plan : PICKUP(A,Init)

Question



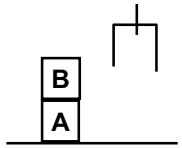
on-table(A,Init)
on(B,A,Init)
clear(B,Init)
arm-empty(Init)

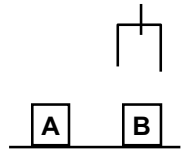


on-table(A,Goal)
clear(A,Goal)
on-table(B,Goal)
clear(B,Goal)
arm-empty(Goal)

Can we derive a plan?

Problem





on-table(A,Init), on(B,A,Init), clear(B,Init), arm-empty(Init)

holds(B,UNSTACK(B,A,Init)) clear(A,UNSTACK(B,A,Init))
¬ on(B,A,UNSTACK(B,A,Init)), ¬ clear(B,UNSTACK(B,A,Init)),
¬ arm-empty(UNSTACK(B,A,Init))

clear(B,PUTDOWN(UNSTACK(B,A,Init))) on-table(B,PUTDOWN(UNSTACK(B,A,Init)))
arm-empty(PUTDOWN(UNSTACK(B,A,Init)))
¬ holds(B, PUTDOWN(UNSTACK(B,A,Init)))

on-table(A,Goal) clear(A,Goal) on-table(B,Goal)
clear(B,Goal) arm-empty(Goal)

Situation Calculus & Frame problem (2)

- Adding axioms to express what remains unchanged, in addition to conditions of applicability and changes in the world
- Example of frame axioms for UNSTACK

$\forall$ (x,y,z, ξ) on-table(z, ξ) $\rightarrow$ on-table(z, UNSTACK(x,y, ξ))

$\forall$ (x,y,u,v, ξ) on(u,v, ξ) $\wedge$ u $\neq$ x$\rightarrow$ on(u,v, UNSTACK(x,y, ξ))

$\forall$ (x,y,z, ξ) clear(z, ξ) $\wedge$ z $\neq$ x$\rightarrow$ clear(z, UNSTACK(x,y, ξ))

« Frame Problem »

- The specification of operators in logic defines the application conditions and changes in the world resulting from the operator's application, BUT NOT what remains unchanged

→ Need to define axioms specifying what remains unchanged (frame axioms)

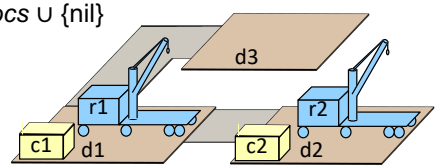
- What characteristics should be included in the definition of actions and states?
  - Example for the Block World:
  - the fact that when a block is moved, the blocks not affected by the move remain in the same state, ...

➡ A solution: use state operators

Outline

State-Variable Representation

- *E* : environment that we want to represent
- *B*: set of objects
  - names for objects in *E*, mathematical constants, …
- Example
  - *B* = *Robots* ∪ *Containers* ∪ *Locs* ∪ {nil}
    - *Robots* = {r1, r2}
    - *Containers* = {c1, c2}
    - *Locs* = {d1, d2, d3}



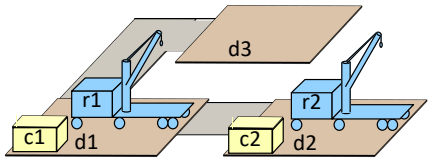
- *B* only needs to include objects that matter at the current level of abstraction
  - can omit lots of details
  - e.g. colors of the robots, the positions of their wheels, …

Properties of Objects

- Define ways to represent properties of objects
  - Two kinds of properties: *rigid* and *varying*

- Rigid property:
  - A property is *rigid* if it stays the same in every state
  - Represented as a mathematical relation

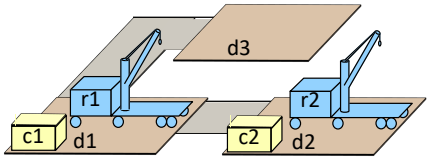
adjacent = {(d1,d2), (d2,d1)}
or
adjacent(d1,d2),
adjacent(d2,d1)



Properties of Objects (contd)

- A property is *varying* if it may differ in different states
  - Represented using a *state variable* that we can assign a value to

- Set of state variables
  X = {loc(r1), loc(r2), loc(c1), loc(c2), cargo(r1), cargo(r2)}

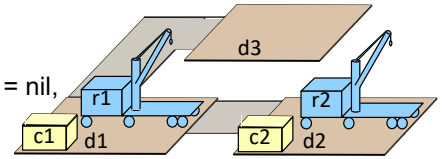


- For each state variable $x \in X$, let Range(x) = {all values that can be assigned to x}
  - Range(loc(r1)) = Range(loc(r2)) = *Locs*
  - Range(loc(c1)) = Range(loc(c2)) = *Robots* ∪ *Locs*
  - Range(cargo(r1)) = Range(cargo(r2)) = *Containers* ∪ {nil}

States as Functions

- Represent each state as a *variable-assignment function*
  - Function that maps each $x \in X$ to a value in Range(x)

s_1(loc(r1)) = d1, s_1(loc(r2)) = d2,
s_1(cargo(r1)) = nil, s_1(cargo(r2)) = 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), F), (loc(c1), d1), …}

- Write it as a set of *ground positive literals* (or *ground atoms*):
  s_1 = {loc(r1)=d1, cargo(r1)=nil, loc(c1)=d1,
  loc(r2)=d2, cargo(r2)=nil, loc(c2)=d2}

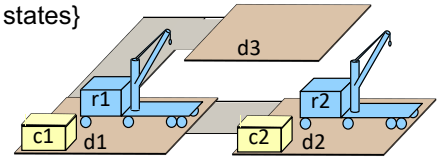
state variable *target*

States as Functions

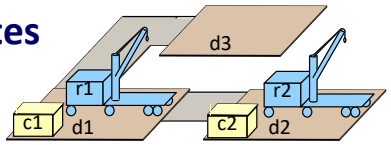
- Let s be a variable-assignment function
  - s is a state only if it has a sensible meaning in E

- Mathematical logic
  - *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 variable property in E

- *State*: a variable-assignment function s such that I(s) can occur in E
  - *State space* S = {all of the states}



Action Templates



- Action template:
  - Parameterized set of actions

  α = (head(α), pre(α), eff(α), cost(α))
- head(α): *name, parameters*

  Each parameter has a range ⊆ *B*, e.g., Range(*r*) = *Robots*
- pre(α): *precondition* literals

  rel(t_1,…,t_k), *sv*(t_1,…,t_k)=t_0,

  ¬*rel*(t_1,…,t_k), *sv*(t_1,…,t_k)≠t_0

  Each t_i is a parameter or element of *B*
- eff(α): *effect* literals

  sv(t_1,…,t_k) ← t_0
- cost(α): *a number*
  - Optional, default value is 1

move(*r,l,m*)
pre: loc(*r*)=*l*, adjacent(*l, m*)
eff: loc(*r*) ← *m*
cost: 1

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

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

Range(*r*) = *Robots*

Range(*l*) = *Robots* ∪ *Locs*

Range(*c*) = *Containers*

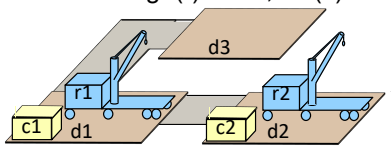
Actions

- A = set of action templates

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

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

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



- Action: *ground instance* of an *α*∈A
  - replace all parameters with values

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

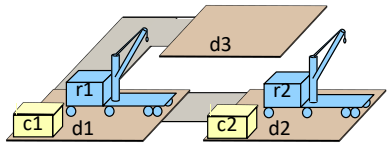
take(r2,d2,c2)
pre: cargo(r2)=nil, loc(r2)=d2, loc(c2)=d2
eff: cargo(r2) ← c2, loc(c1) ← r2

put(r1,d1,c1)
pre: loc(r1)=d1, loc(c1)=r1
eff: cargo(r1) ← c1, loc(c1) ← d1

A = {all actions we can get from A}
= {all ground instances of members of A }

Applicability

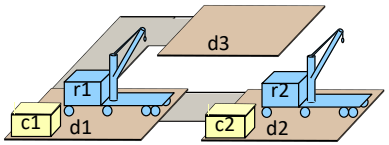
- *a* is *applicable* in *s* if
  - for every positive literal *l* ∈ pre(*a*), *l* is in *s* or in one of the rigid relations
  - for every negative literal ¬*l* ∈ pre(*a*) *l* is not in *s,* nor in any of the rigid relations
- s_1 = {cargo(r1)=nil, cargo(r2)=nil, loc(r1)=d1, loc(r2)=d2, loc(c1)=d1, loc(c2)=d2}



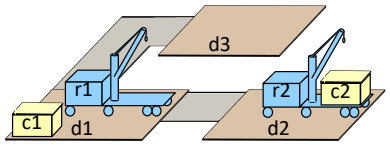
- take(r2,d2,c2)
  pre: cargo(r2)=nil, loc(r2)=d2, loc(c2)=d2
  eff: cargo(r2) ← c2, loc(c1) ← r2

Computing *γ*

- *γ*(*s,a*) = {(*x,w*) | eff(*a*) contains the effect *x* ← *w*}
  ∪ {(*x,w*) ∈ *s* | *x* isn't the target of any effect in eff(*a*)}
- s_1 = {cargo(r1)=nil, cargo(r2)=nil, loc(r1)=d1, loc(r2)=d2, loc(c1)=d1, loc(c2)=d2}



- take(r2,d2,c2)
  pre: cargo(r2)=nil, loc(r2)=d2, loc(c2)=d2
  eff: cargo(r2) ← c2, loc(c1) ← r2
- γ(s_1, take(r2,d2,c2)) =
  {cargo(r1)=nil, cargo(r2)=**c2**, loc(r1)=d1, loc(r2)=d2, loc(c1)=d1, loc(c2)=**r2**}



State-Variable Planning Domain

- Let

  B = finite set of objects

  R = finite set of rigid relations over B

  X = finite set of state variables
  - for every state variable x, Range(x) $\subseteq B$

  S = state space over X

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

  A = finite set of action templates
  - for every parameter y, Range(y) $\subseteq B$

  A = {all ground instances of action templates in A}

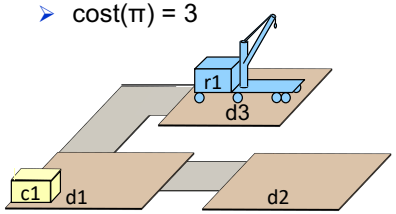
  $\gamma(s,a)$ = {(x,w) | eff(a) contains the effect $x \leftarrow w$}
  $\cup$ {(x,w)$\in s$ | x isn't the target of any effect in eff(a)}

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

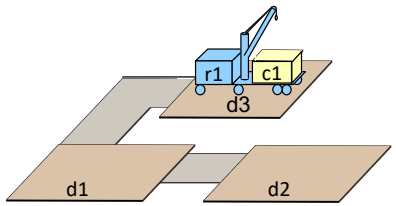
Plans

- *Plan*: sequence of actions $\pi = \langle a_1,a_2,\ldots,a_n\rangle$
  - cost(π) = $\sum_i$ cost(a_i)
- π is *applicable* in s_0 if the actions can be applied in the order given
  - $\gamma(s_0,a_1) = s_1$, $\gamma(s_1,a_2) = s_2$, …, $\gamma(s_{n-1},a_n) = s_n$
  - Define $\gamma(s_0, \pi) = s_n$

  - π = ⟨move(r2,d3,d1), take(r1,d1,c1), move(r1,d1,d3)⟩
  - cost(π) = 3



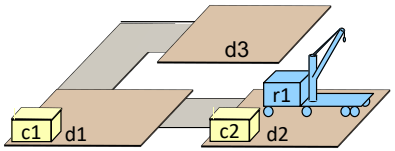
s_0 = {loc(r1)=d3, cargo(r1)=nil, loc(c1)=d1}



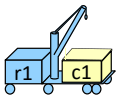
$\gamma(s_0,\pi)$ = {loc(r1)=d3, cargo(r1)=**c1**, loc(c1)=**r1**}

Planning Problems

- *State-variable planning problem*: a triple $P = (\Sigma,s_0,g)$
  - $\Sigma = (S,A,\gamma)$ 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 = {$s\in S$ | every positive literal in g is also in s or R, and none of the negative literals in g are in s or R}



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



g = {cargo(r1)=c1}

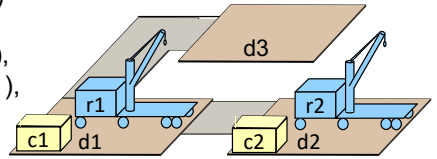
- This is a *state-variable representation* of a *classical planning problem*
  - There's also *classical representation* …

Outline

Classical Representation

- Motivation
  - The field of AI planning started out as automated theorem proving
  - *Based on STRIPS (Fikes, Nilsson 1971)*
- Classical representation is equivalent to state-variable representation
  - Represents both rigid and varying properties using logical predicates
    - adjacent(*l,m*) - location *l* is adjacent to location *m*
    - loc(*r,l*) - robot *r* is at location *l*
    - loc(*c,l*), loc(*c,r*) - container *c* is at location *l* or on robot *r*
    - cargo(*r*) - there is a container on robot *r*
- State *s* = a set of ground atoms (Closed world assumption: what is not currently known to be true, is false)

s_1 = {adjacent(d1,d2), adjacent(d2,d1),
adjacent(d1,d3), adjacent(d3,d1),
loc(c1,d1), loc(c2,d2),
loc(r1,d1), cargo(r1)=nil,
loc(r2,d2), cargo(r2)=nil}



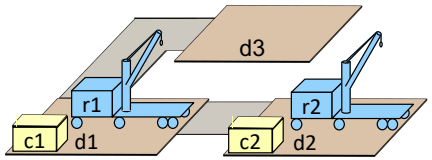
Classical Planning Operators

- *o* = (head(*o*), pre(*o*), eff(*o*))
- Each precondition and effect:
  pred(t_1,…,t_k)
  ¬*pred*(t_1,…,t_k)
  - *pred* = predicate name
  - each t_i must be a constant (i.e., member of *B*) or a parameter
- Action: a ground instance

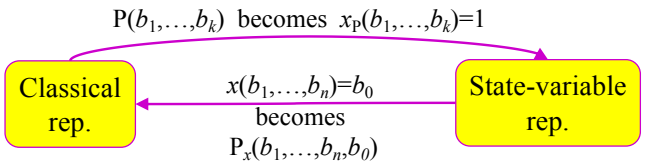
move(*r,l,m*)
Precond: loc(*r,l*),
Effects: ¬loc(*r,l*), loc(*r,m*)

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

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



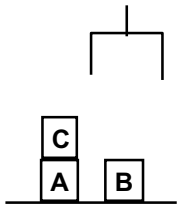
Discussion



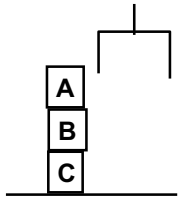
- Equivalent expressive power
  - Each can be converted to the other in linear time and space
- Classical representation is more natural for logicians
- State-variables are more natural for most engineers and computer scientists
  - When changing a value, you don't have to explicitly delete the old one
- Historically, classical representation has been more widely used
  - That's starting to change

Example / States

- Predicates :
  - on-table/1, arm-empty/0, holding/1, on/2



Initial state
on-table(B)
on-table(A)
on(C,A)
clear(C)
clear(B)
arm-empty



Goal state
on(A,B)
on(B,C)
on-table(C)
arm-empty

Example: operators

- Pickup(o):
  Precond: clear(o), on-table(o), arm-empty
  Effect: ¬ on-table(o), ¬ clear(o), ¬ arm-empty, holding(o)
- Putdown(o):
  Precond: holding(o)
  Effect: ¬ holding(o), clear(o), arm-empty(), on-table(o)
- Stack(o,uo):
  Precond: clear(uo), holding(o)
  Effect: ¬ holding(o), ¬ clear(uo), clear(o), arm-empty(), on(o,uo)
- Unstack(o,uo);
  Precond: on(o,uo), clear(o), arm-empty()
  Effect: ¬ on(o,uo), ¬ clear(o), ¬ arm-empty(), holding(o), clear(uo)

QUESTION

- Simple planning problem:
  - Two **crates**
    - At A
    - Should be at B

  - One **robot**
    - Can **carry** up to two crates
    - Can **move** between locations, which requires one unit of **fuel**
    - Has only two units of fuel

Domain modeling

- State modeling
  - robotAt/1, at/2, holding/1 (= carrying)
  - has-fuel/1, capacity/1
- Action modeling
  - pickup(X,L)
    - Precond: robotAt(L), capacity(C), C<2, at(X,L)
    - Effect: capacity(C+1), holding(X), not at(X,L)
  - put(X,L)
    - Precond: robotAt(L), holding(X), capacity(C), C>=0
    - Effect: capacity(C-1),not holding(X),at(X,L)
  - drive(L1,L2)
    - Precond: robotAt(L1), has-fuel(Q), Q >=1
    - Effect: has-fuel(Q-1), not robotAt(L1), robotAt(L2)

Outline

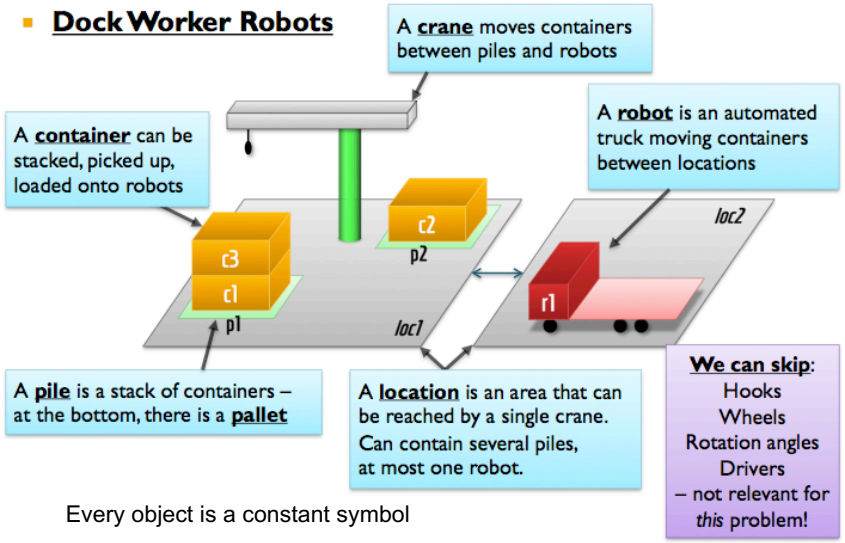
PPDL

- Standard encoding language for “classical” planning tasks
- Components of a PDDL planning task definition:
  - **Objects:** Things in the world that interest us.
  - **Predicates:** Properties of objects that we are interested in; can be *true* or *false*.
  - **Initial state:** The state of the world that we start in.
  - **Goal specification:** Things that we want to be true.
  - **Actions/Operators:** Ways of changing the state of the world.
- Planning tasks defined in PDDL are separated into two files:
  - A domain file for predicates and actions.
  - A problem file for objects, initial state and goal specification.

- BNF PDDL 3.0 *www.cs.yale.edu/homes/dvm/papers/pddl-**bnf**.**pdf***
- DWR Example http://www.laas.fr/planning/DWR-operators

Finite set of objects relevant to the problem

- **Dock Worker Robots**



Every object is a constant symbol

Predicates

- **Properties** of the **world**
  - **raining** – it is raining [not part of the DWR domain!]
- **Properties** of single **objects**…
  - **occupied**(*robot*) – the robot has a container
- **Relations** between objects
  - **attached**(*pile*, *location*) – the pile is in the given location
- **Relations** between >2 objects
  - **can-move**(*robot, loc, loc*) – the robot can move between two locations
- **Non-boolean properties** are "relations between constants"
  - **has-color**(*robot, color*) – the robot has the given color

Determine what is relevant to the problem and objective

Predicates

"Fixed/Rigid" (can't change)

adjacent	(*loc1*, *loc2*)	; can move from *loc1* directly to *loc2*
attached	(*p*, *loc*)	; pile *p* attached to *loc*
belong	(*k*, *loc*)	; crane *k* belongs to *loc*

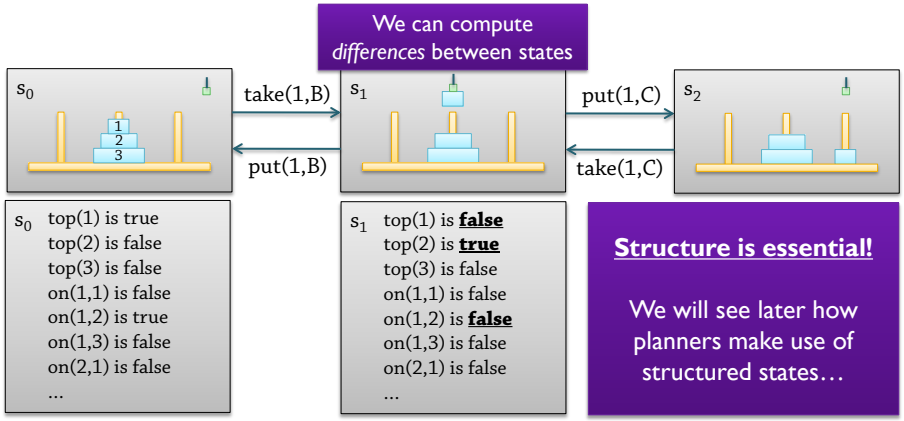
"Dynamic" (modified by actions)

at	(*r*, *loc*)	; robot *r* is at *loc*
occupied	(*loc*)	; there is a robot at *loc*
loaded	(*r*, *c*)	; robot *r* is loaded with container *c*
unloaded	(*r*)	; robot *r* is empty
holding	(*k*, *c*)	; crane *k* is holding container *c*
empty	(*k*)	; crane *k* is not holding anything
in	(*c*, *p*)	; container *c* is somewhere in pile *p*
top	(*c*, *p*)	; container *c* is on top of pile *p*
on	(*c1*, *c2*)	; container *c1* is on container *c2*

Term: Constant symbol or variable
Atom: Predicate symbol applied to the intended number of terms
Ground atom: Atom without variables (only constants)

States

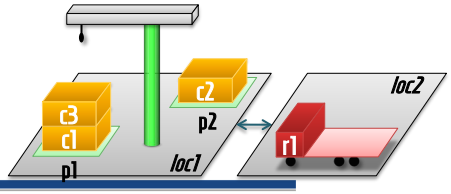
A state **is** a **set** of all **ground atoms** that are true



Operators

- In the classical representation: Don't define actions directly
  - Define a set O of operators
  - Each ***operator*** is parameterized, defines many actions
    - ;; crane k at location l takes container c off container d in pile p
      take(k, l, c, d, p)
  - Has a **precondition**
    - precond(o): **set** of **literals** that must hold before execution
    - precond(take) = { belong(k,l), empty(k), attached(p,l), top(c,p), on(c,d) }
  - Has **effects**
    - effects(o): **set** of **literals** that will be made to hold after execution
    - effects(take) = { holding(k,c), ¬empty(k), ¬in(c,p), ¬top(c,p), ¬on(c,d), top(d,p) }

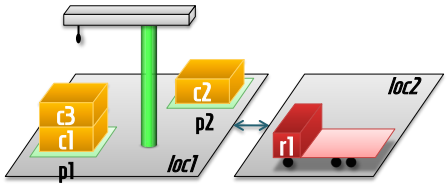
Preconditions should have to do with *executability*, not *suitability*
They are the weakest constraints under which the action *can* be executed



Actions

- In the classical representation:
  - Every **ground instantiation** of an operator is an **action**
    - a_1 = take(crane1, loc2, c3, c1, p1)
  - Also has (instantiated) precondition, effects
    - precond(a_1) = { belong(crane1,loc2), empty(crane1), attached(p1,loc2), top(c3,p1), on(c3,c1) }
    - effects(a_1) = { holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1), top(c1,p1) }

$$A = \left\{ a \;\middle|\; \begin{array}{l} a \text{ is an instantiation} \\ \text{of an operator in } O \\ \text{using constants in } L \end{array} \right\}$$

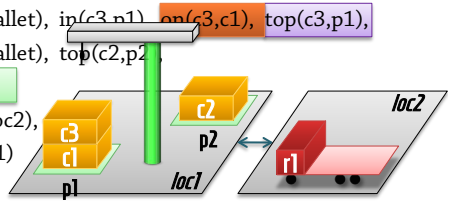


Applicable Actions

- An action a is **applicable** in a state s…
  - … if precond+(a) ⊆ s and precond–(a) ∩ s = ∅
- Example:
  - take(crane1, loc1, c3, c1, p1):
    ;; crane1 at loc1 takes c3 off c1 in pile p1
    precond: { belong(crane1,loc1), empty(crane1), attached(p1,loc1), top(c3,p1), on(c3,c1) }
    effects: { holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1), top(c1,p1) }
  - s1 = {
    attached(p1,loc1), in(c1,p1), on(c1,pallet), in(c3,p1), on(c3,c1), top(c3,p1),
    attached(p2,loc1), in(c2,p2), on(c2,pallet), top(c2,p2),
    belong(crane1,loc1), empty(crane1),
    at(r1,loc2), unloaded(r1), occupied(loc2),
    adjacent(loc1,loc2), adjacent(loc2,loc1)
    }

Action → ground → preconds are **ground atoms**

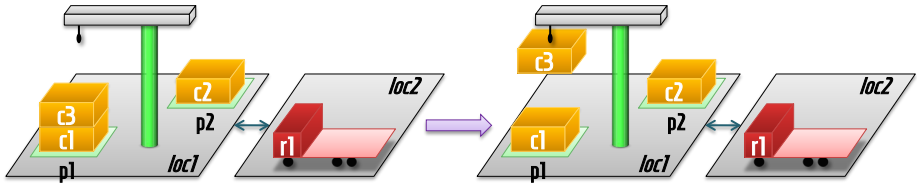
Simple representation (sets) → simple definitions!



Applying Actions

- **Applying** will **add** positive effects, **delete** negative effects
  - If a is applicable in s, then
    the new state is $(s - \text{effects}^-(a)) \cup \text{effects}^+(a)$

  - **take**(crane1, loc1, c3, c1, p1):
    ;; crane1 at loc1 takes c3 off c1 in pile p1
    precond: belong(crane1,loc1), empty(crane1),
    attached(p1,loc1), top(c3,p1), on(c3,c1)
    effects: holding(crane1,c3), top(c1,p1),
    ¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1)



PDDL Introduction

- A PDDL definition consists of two parts: *domain* and *problem* definition.
  - Although not required by the PDDL standard, most planners require that the two parts are in separate files.
- Comments in a PDDL file start with a semicolon (";") and last to the end of the line.
- Requirements: because PDDL is a very general language and most planners support only a subset, domains may declare requirements.
- The most commonly used requirements are:
  - :**strips** The most basic subset of PDDL, consisting of STRIPS only
  - :**equality** This requirement means that the domain uses the predicate =, interpreted as equality.
  - :**typing** This requirement means that the domain uses types
  - :**adl** Means that the domain uses some or all of ADL (*i.e.* disjunctions and quantifiers in preconditions and goals, quantified and conditional effects).

PDDL: Domain and Problem Definition

```
(define (domain <domain name>)
   <PDDL code for predicates>
   <PDDL code for first action>
   [...]
   <PDDL code for last action>
)

(define (problem <problem name>)
   (:domain <domain name>)
   <PDDL code for objects>
   <PDDL code for initial state>
   <PDDL code for goal specification>
)
```


PDDL Domain Definition

- Contains the domain predicates and operators (called actions in PDDL) and may also contain types, constants, static facts, ….

```
(define (domain DOMAIN_NAME)
      (:requirements [:strips] [:equality] [:typing] [:adl])
      (:predicates (PREDICATE_1_NAME ?A1 ?A2 ... ?AN) ...)
      (:action ACTION_1_NAME
          [:parameters (?P1 ?P2 ... ?PN)]
          [:precondition CONDITION_FORMULA]
          [:effect EFFECT_FORMULA] )
      (:action …)
   ...)
```

PDDL Domain Definition

- Elements in []'s are optional,
- Names (domain, predicate, action, et c.) are usually made up of alphanumeric characters, hyphens ("-") and underscores ("_"), but there may be some planners that allow less.
- Parameters of predicates and actions are distinguished by their beginning with a question mark ("?")
  - in predicate declarations (the :predicates part) parameters have no other function than to specify the number of arguments that the predicate should have, i.e. the parameter names do not matter (as long as they are distinct).
- Predicates can have zero parameters (but in this case, the predicate name still has to be written within parenthesizes).

PDDL Problem Definition

- The problem definition contains the objects present in the problem instance, the initial state description and the goal.

```
(define (problem PROBLEM_NAME)
    (:domain DOMAIN_NAME)
    (:objects OBJ1 OBJ2 ... OBJ_N)
    (:init (and ATOM1 ATOM2 ... ATOM_N))
    (:goal CONDITION_FORMULA)
)
```

A precondition formula may be an atomic formula or a conjunction of atomic formulas: (and ATOM1 ... ATOM_N)

Question: Gripper example

- A robot that can move between two rooms and pick up or drop balls with either of his two arms.
- Initially, four balls and the robot are in a first room.
- We want the balls to be in the second room.
- **Objects:** The two rooms, four balls and two robot arms.
- **Predicates:** Is x a room? Is x a ball? Is ball x inside room y? Is robot arm x empty? …
- **Initial state:** All balls and the robot are in the first room. All robot arms are empty.
- **Goal specification:** All balls must be in the second room.
- **Actions/Operators:** The robot can move between rooms, pick up a ball or drop a ball.

Gripper example

Predicates:

`ROOM(`x`)`	– true iff x is a room
`BALL(`x`)`	– true iff x is a ball
`GRIPPER(`x`)`	– true iff x is a gripper (robot arm)
`at-robby(`x`)`	– true iff x is a room and the robot is in x
`at-ball(`x`, `y`)`	– true iff x is a ball, y is a room, and x is in y
`free(`x`)`	– true iff x is a gripper and x does not hold a ball
`carry(`x`, `y`)`	– true iff x is a gripper, y is a ball, and x holds y

In PDDL:

```
(:predicates (ROOM ?x) (BALL ?x) (GRIPPER ?x)
             (at-robby ?x) (at-ball ?x ?y)
             (free ?x) (carry ?x ?y))
```

Gripper example

Action/Operator:

Description: The robot can move from x to y.
Precondition: `ROOM(`x`)`, `ROOM(`y`)` and `at-robby(`x`)` are true.
Effect: `at-robby(`y`)` becomes true. `at-robby(`x`)` becomes false. Everything else doesn't change.

In PDDL:

```
(:action move :parameters (?x ?y)
   :precondition (and (ROOM ?x) (ROOM ?y)
                      (at-robby ?x))
   :effect       (and (at-robby ?y)
                      (not (at-robby ?x))))
```


Gripper example

Action/Operator:

Description: The robot can pick up x in y with z.
Precondition: `BALL(`x`)`, `ROOM(`y`)`, `GRIPPER(`z`)`, `at-ball(`x`, `y`)`, `at-robby(`y`)` and `free(`z`)` are true.
Effect: `carry(`z`, `x`)` becomes true. `at-ball(`x`, `y`)` and `free(`z`)` become false. Everything else doesn't change.

In PDDL:

```
(:action pick-up :parameters (?x ?y ?z)
   :precondition (and (BALL ?x) (ROOM ?y) (GRIPPER ?z)
                      (at-ball ?x ?y) (at-robby ?y) (free ?z))
   :effect       (and (carry ?z ?x)
                      (not (at-ball ?x ?y)) (not (free ?z))))
```


Gripper example

Action/Operator:

Description: The robot can drop x in y from z.

(Preconditions and effects similar to the pick-up operator.)

In PDDL:

```
(:action drop :parameters (?x ?y ?z)
   :precondition (and (BALL ?x) (ROOM ?y) (GRIPPER ?z)
                      (carry ?z ?x) (at-robby ?y))
   :effect       (and (at-ball ?x ?y) (free ?z)
                      (not (carry ?z ?x))))
```


Problem definition

- Objects
- Initial state
- Goal state

Gripper Example

Objects:

Rooms: `rooma`, `roomb`
Balls: `ball1`, `ball2`, `ball3`, `ball4`
Robot arms: `left`, `right`

In PDDL:

```
(:objects rooma roomb
          ball1 ball2 ball3 ball4
          left right)
```


Gripper example

Initial state:

`ROOM(rooma)` and `ROOM(roomb)` are true.
`BALL(ball1)`, ..., `BALL(ball4)` are true.
`GRIPPER(left)`, `GRIPPER(right)`, `free(left)` and `free(right)` are true.
`at-robby(rooma)`, `at-ball(ball1, rooma)`, ..., `at-ball(ball4, rooma)` are true.
Everything else is false.

In PDDL:

```
(:init (ROOM rooma) (ROOM roomb)
       (BALL ball1) (BALL ball2) (BALL ball3) (BALL ball4)
       (GRIPPER left) (GRIPPER right) (free left) (free right)
       (at-robby rooma)
       (at-ball ball1 rooma) (at-ball ball2 rooma)
       (at-ball ball3 rooma) (at-ball ball4 rooma))
```


Gripper example

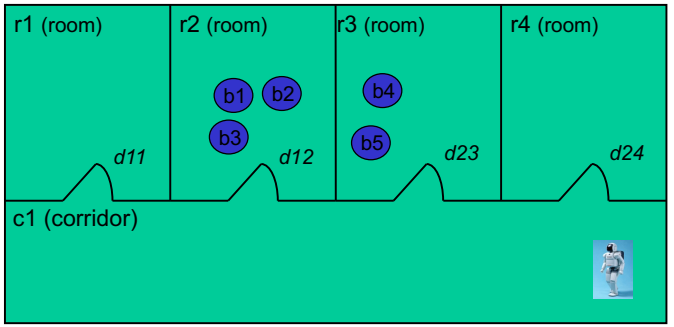
Goal specification:

`at-ball(ball1, roomb)`, ..., `at-ball(ball4, roomb)` must be true.
Everything else we don't care about.

In PDDL:

```
(:goal (and (at-ball ball1 roomb)
            (at-ball ball2 roomb)
            (at-ball ball3 roomb)
            (at-ball ball4 roomb)))
```


Question: Gripper example revisited



After IFT702, Froduald Kabanza

Gripper example revisited
Domain definition

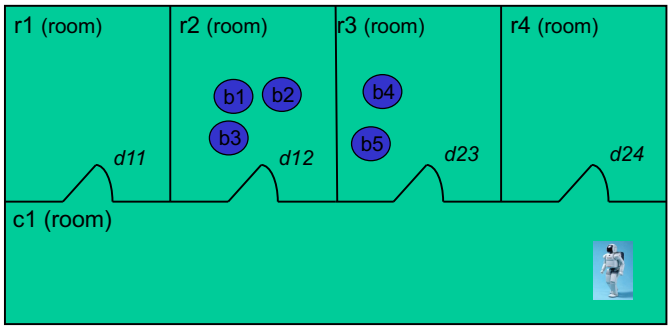
```
(define (domain robotWorld1)
  (:types gripper ball room door)
  (:predicates (atRobot ?x - room) (at ?b - ball ?x - romm )
               (free ?g - gripper) (holding ?g - gripper ?b - ball)
               (connects ?d - door ?r1 - room ?r2 - room))

  (:action pick
     :parameters (?b - ball ?g - gripper ?r - room)
     :precondition (and (at ?b ?r) (atRobot ?r) (free ?g))
     :effect (and (holding ?g ?b) (not (at ?b ?r)) (not (free ?g))))

  (:action release
   :parameters (?b - ball ?g - gripper ?r - room)
   :precondition (and (holding ?g ?b) (atRobot ?r))
   :effect (and (at ?b ?r) (not (holding ?g ?b)) (free ?g)))

  (:action move
   :parameters (?rf ?rt - room ?d - door)
   :precondition (atRobot ?rf) (connects ?d ?rf ?rt)
   :effect (and (atRobot ?rt) (not (atRobot ?rf))))
```


Gripper example revisited
Problem definition




```
(:objects b1 b2 b3 b4 b5 - ball  left right - gripper
          r1 r2 r3 r4 c1 - room
           d11 d12 d23 d24 - door)
(:init (atRobot c2) (free left) (free right)
       (at b1 r2) (at b2 r2) (at b3 r2) (at b4 r3) (at b5 r3)
        (connects d11 r1 c1) (connects d12 r2 c1)
        (connects d23 r3 c1) (connects d24 r4 c1))
(:goal (at b1 r4) (at b2 r4) (at b3 r4) (at b4 r4) (at b5 r4))
  ; OR(:goal (forall (?x - ball)(at ?x r4))
)
```


Question: Cargo Domain Definition

(define (domain cargo) (:requirements :strips)

(:predicates (in ?c ?p) (out ?c) (at-airport ?x ?a)
(plane ?p) (airport ?a) (cargo ?c))

(:action load :parameters (?c ?p ?a)
:precondition (and (at-airport ?c ?a) (at-airport ?p ?a) (out ?c)
(plane ?p) (cargo ?c) (airport ?a))
:effect (and (not (at-airport ?c ?a)) (in ?c ?p) (not (out ?c))))

- (:action unload :parameters (?c ?p ?a)
:precondition (and (in ?c ?p) (at-airport ?p ?a)
(plane ?p) (cargo ?c) (airport ?a))
:effect (and (not (in ?c ?p)) (at-airport ?c ?a) (out ?c)))

(:action fly :parameters (?p ?from ?to)

:precondition (and (at-airport ?p ?from)
(plane ?p) (airport ?from) (airport ?to))

- :effect (and (not (at-airport ?p ?from)) (at-airport ?p ?to))))
-

Question: Cargo Problem Definition

(define (problem cargo1) (:domain cargo)
(:objects c1 c2 jfk sfo p1 p2)

(:init (airport jfk) (airport sfo)
(plane p1) (plane p2)
(at-airport p1 jfk) (at-airport p2 sfo)

(cargo c1) (cargo c2)
(at-airport c1 sfo) (at-airport c2 jfk)
(in c2 p1) (out c1))

(:goal (and (at-airport c1 jfk) (out c1) (at-airport c2 sfo) (out c2))))

PDDL Example for Blocks World

```
(:action unstack
  :parameters (?x – block ?y – block)
  :precondition  (and (on ?x ?y) (clear ?x) (handempty)
  :effects  (and (not  (on ?x ?y)) (not (clear ?x))
                 (not (handempty)) (holding ?x) (clear ?y))
(:action stack
   :parameters (?x – block ?y – block)
   :precondition (and (holding ?x) (clear ?y))
   :effects (and (not (holding ?x)) (not (clear ?y))
                 (on ?x ?y) (clear ?x) (handempty))
(:action pickup
  :parameters (?x – block)
  :precondition  (and (ontable ?x) (clear ?x) (handempty)
  :effects   (and (ontable ?x) (clear ?x) (handempty) (holding ?x))
(:action putdown
:parameters (?x – block)
 :precondition (holding ?x)
 :effects  (and (not (holding ?x)) (ontable ?x) (clear ?x) (handempty))
```


Question: Rubik's Cube

- 1Dimensional Rubik's Cube is a line of 6 numbers with original position: 1 2 3 4 5 6
- 3 different ways of rotating it:
  - (1 2 3 4) 5 6 → (4 3 2 1) 5 6
  - 1 (2 3 4 5) 6 → 1 (5 4 3 2) 6
  - 1 2 (3 4 5 6) → 1 2 (6 5 4 3)

Rubik's Cube

```
(define (domain rubikscube)
    (:requirements :strips)
    (:predicates (oncase1  ?y ) (oncase2  ?y ) (oncase3  ?y )
                 (oncase4  ?y ) (oncase5  ?y ) (oncase6  ?y ))
    (:action MoveOne :parameters (?x  ?y  ?z  ?t)
      :precondition (and (oncase1 ?x ) (oncase2 ?y) ( oncase3 ?z)
                         (oncase4 ?t))
      :effect (and (oncase1 ?t) (oncase2 ?z) (oncase3 ?y) (oncase4 ?x)
                   (not(oncase1 ?x)) (not(oncase2 ?y))
                   (not(oncase3 ?z)) (not(oncase4 ?t))))
    (:action MoveTwo :parameters (?x  ?y  ?z  ?t)
          :precondition (and (oncase2 ?x ) (oncase3 ?y)
                         (oncase4 ?z) (oncase5 ?t))
          :effect (and (oncase2 ?t) (oncase3 ?z) (oncase4 ?y)
                   (oncase5 ?x) (not (oncase2 ?x)) (not (oncase3 ?y))
                   (not (oncase4 ?z)) (not (oncase5 ?t))))
```



```
(:action MoveThree :parameters (?x  ?y  ?z  ?t)
    :precondition (and (oncase3 ?x ) (oncase4 ?y)
                       ( oncase5 ?z) (oncase6 ?t))
    :effect (and (oncase3 ?t) (oncase4 ?z) (oncase5 ?y)
                 (oncase6 ?x) (not(oncase3 ?x))
                 (not(oncase4 ?y)) (not(oncase5 ?z))
                 (not(oncase6 ?t)))))
```