Course "Automated Planning: Theory and Practice"
Chapter 02: Classical Planning and PDDL

Teacher:	Marco Roveri - `marco.roveri@unitn.it`
M.S. Course:	Artificial Intelligence Systems (LM)
A.A.:	2023-2024
Where:	DISI, University of Trento
URL:	`https://bit.ly/3zOkGk8`

Last updated: Sunday 17th September, 2023

Terms of Use and Copyright

The material (text, figures) in these slides is authored by Jonas Kvarnström and Marco Roveri.

HISTORY: AROUND 1959

- The language of *Artificial Intelligence* was/is logic
  - First-order, second-order, modal, ...
- 1959: General Problem Solver (GPS) Newell et al. [11]
  - General Problem Solver (GPS) is a computer program created in 1959 by Herbert A. Simon, J. C. Shaw, and Allen Newell (RAND Corporation) intended to work as a universal problem solver machine.[a]

REPORT ON A GENERAL PROBLEM-SOLVING PROGRAM

This paper deals with the theory of problem solving. It describes a program for a digital computer, called General Problem Solver I (GPS), which is part of an investigation into the extremely complex processes that are involved in intelligent, adaptive, and creative behavior. Our principal means of investigation is synthesis: programming large digital computers to exhibit intelligent behavior, studying the structure of these computer programs, and examining the problem-solving and other adaptive behaviors that the programs produce.

a

SUMMARY

This paper reports on a computer program, called GPS-I for General Problem Solving Program I. Construction and investigation of this program is part of a research effort by the authors to understand the information processes that underlie human intellectual, adaptive, and creative abilities. The approach is synthetic — to construct computer programs that can solve problems requiring intelligence and adaptation, and to discover which varieties of these programs can be matched to data on human problem solving.

GPS-I grew out of an earlier program, the Logic Theorist, which discovers proofs to theorems in the sentential calculus. GPS-I is an attempt to fit the recorded behavior of college students trying to discover proofs. The purpose of this

b

[a] https://en.wikipedia.org/wiki/General_Problem_Solver

[b] ftp://bitsavers.informatik.uni-stuttgart.de/pdf/rand/ipl/P-1584_Report_On_A_General_Problem-Solving_Program_Feb59.pdf

History: around 1969

- 1969: planner explicitly built on Theorem Proving Green [8]

Abstract

This paper shows how an extension of the resolution proof procedure can be used to construct problem solutions. The extended proof procedure can solve problems involving state transformations. The paper explores several alternate problem representations and provides a discussion of solutions to sample problems including the "Monkey and Bananas" puzzle and the 'Tower of Hanoi" puzzle. The paper exhibits solutions to these problems obtained by QA3, a computer program bused on these theorem-proving methods. In addition, the paper shows how QA3 can write simple computer programs and can solve practical problems for a simple robot. [a]

[a] https://www.ijcai.org/Proceedings/69/Papers/023.pdf

Basic in logic

- Full theorem proving generally proved impractical for planning
  - Different techniques were found
  - Foundations in logical languages remained!
    - Languages use predicates, atoms, literals, formulas
    - We define states, actions, ... relative to these
    - $\Longrightarrow$ Allows us to specify an STS at a higher level!

Formal Representation using a first-order language

"Classical Representation" (from Ghallab et al. [6])

"The *simplest* representation that is (more or less) reasonable to use for modeling"

Running Example: Dock Worker Robot (DWR)

Containers shipped in/out of an harbor

Cranes move containers between "piles" and robotic trucks

Objects

- We are interested in objects in the world
  - Buildings, cards, aircraft, people, trucks, robots, cranes, crates, ...
  - Classical $\Longrightarrow$ must be a finite set!

Modeling issue

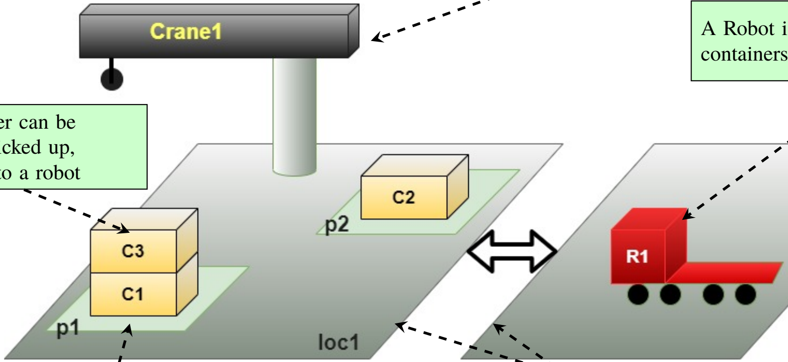
Which objects exist and are relevant for the problem and objectives?

Objects: Dock Worker Robots

A crane moves containers between piles and robots

A Robot is an automated truck moving containers bewteen locations

A container can be stacked, picked up, loaded onto a robot



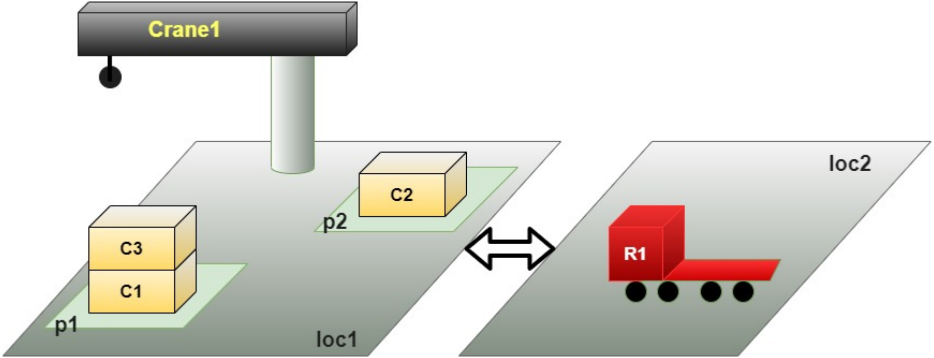
We can skip: Hooks, Wheels, Rotation angles, Drivers,... not relevant for this problem!

A pile is a stack of containers - at the bottom there is a pallet

A location is an area that can be reached by a single crane. Can contain several piles, at most one robot

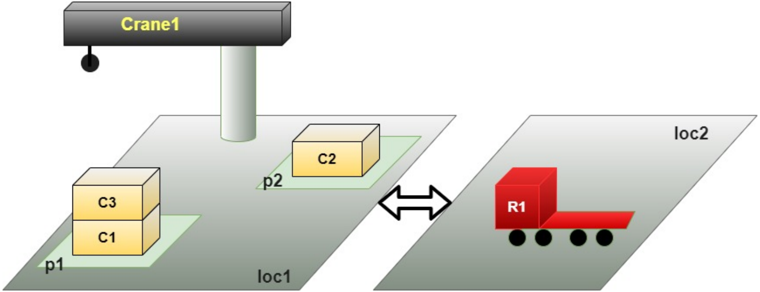
Objects: Classical representation

- We are constructing a first-order language L (as in Logic)
- Every object is modeled as a constant
  - We have a constant symbol ("object name") for each object
  - L contains : { $c1$, $c2$, $c3$, $p1$, $p2$, $loc1$, $loc2$, $r1$, $crane1$, ...}



Predicates, Atoms, States

- An STS only assumes there are states!
  - What is a state? The STS does not care!
  - Its definition do not depend on what s "represents" or "means"!
    - Can execute a in s if $\gamma(s, a) = \{s'\}$
- Planners need more structure!
  - state $s_{1234567900} \Longrightarrow$ "the state where c1 is on c3 on p1 in loc1, c2 is on p2 in loc1, r1 is empty in loc2"



Predicates, Terms, Atoms, Ground Atoms

- 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*
  - can-move(robot,location,location) *- 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** for the **problem** and **objective**!

PREDICATES FOR DWR

"Fixed/Rigid" (can't change)	**adjacent**	*(loc1, loc2)*	*; can move from loc1 to loc2*
	attached	*(p, loc)*	*; pile p attached 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 on pile p*
	top	*(c,p)*	*; container c is on top of pile p*
	on	*(c1,c2)*	*; container c1 is on container c2*

Predicates, Terms, Atoms, Ground Atoms

- Term: Constant symbol or variable
  - loc2 – *constant*
  - location – *variable*
- Atom: Predicate symbol applied to the intended number of terms
  - raining
  - occupied(location)
  - at(r1, loc2)
- Ground atoms: Atom without variables (only constants) - fact
  - occupied(loc2)

- Plain first-order logic has no distinct types for objects!
  - $\Longrightarrow$ Some "strange" atoms are perfectly valid:
    - at(loc1, loc2)
    - holding(loc1, c1)
    - ...

States

- A state (of the world) should specify exactly which facts (ground atoms) are true/false in the world at a given time instant!

We know all predicates that exist:
adjacent(location,location),...

We know which objects exist

$\Longrightarrow$

We can calculate all ground atoms

adjacent(loc1, loc1)
adjacent(loc1, loc2)

...
attached(p1, loc1)
...

These are the facts to keep track of!

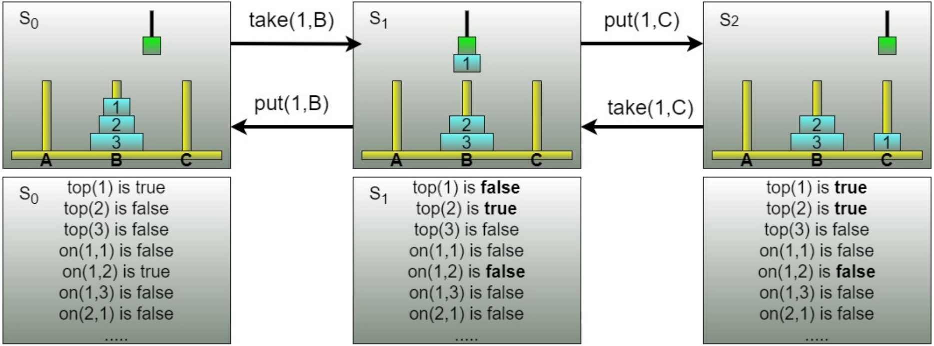
$\Downarrow$

We can find all possible states!

Every assignment of true/false to the ground atoms is a distinct state
Number of states $2^{\text{number of atoms}}$ - enormous, but finite (for classical planning)

STATES: FIRST-ORDER REPRESENTATION

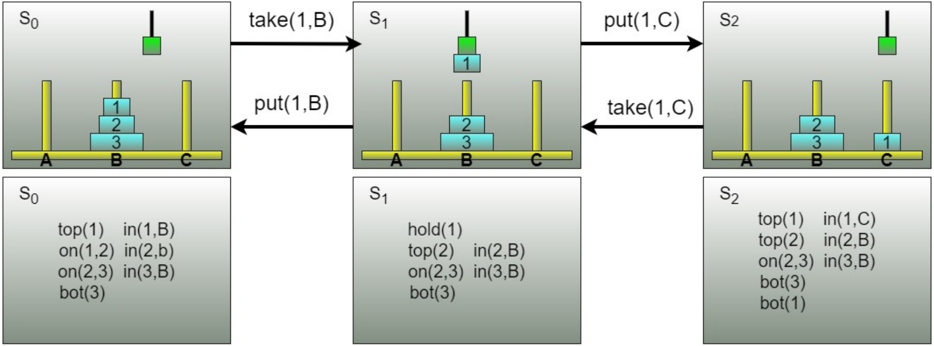
- Then we can compute differences between states!



States: first-order representation

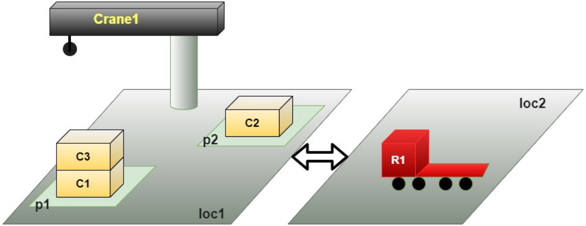
- Efficient specification/storage of a single state
  - Specify which facts are true
    - All other facts have to be false - what else would they be?
  - $\Longrightarrow$ A classical state is a set of all ground atoms that are true
    - $s_0 = \{top(1), on(1,2), on(2,3), in(1,B), in(2,B), in(3,B), bot(3)\}$

$top(1) \in s_0 \rightarrow top(1)$ *is true*
$top(2) \notin s_0 \rightarrow top(2)$ *is false*



States: Initial state

- Initial state in classical planning
  - We assume a complete information about the initial state s_0 (and of any state before any action)
  - State = set of true facts...
    - $s_0 = \{attached(p1, loc1), in(c1, p1), on(c1, pallet), on(c3, p1), \ldots\}$



Complete relative to the model

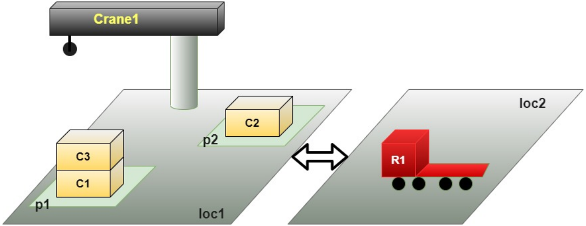
We must know everything about those predicates and objects we have specified...

States: Goal states

- A goal g is a finite set of ground atoms
  - Example: In the final state, containers $c1$ and $c2$ should be on pile $p2$ and we do not care about the other facts
    - $g = \{in(c1, p2), in(c3, p2)\}$
- Thus, $S_g = \{s \in S | g \subseteq s\}$
  - $S_g = \{$ {in(c1,p2), in(c3,p2)} *– one acceptable final state*

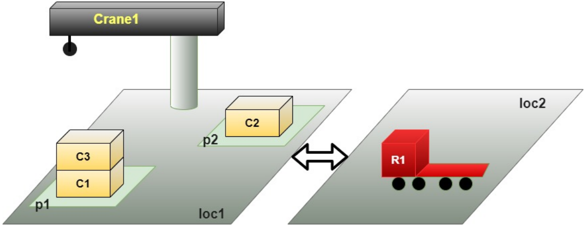
    {in(c1,p2), in(c3,p2), on(c1,c3)} *– another acceptable final state*

    ... $\}$



States: Goal states (alt. definition)

- A goal g is a set of ground literals
  - A literal is an atom or a *negated* atom: $in(c1, p2), \neg in(c2, p3)$
    - $in(c1, p2) \Longrightarrow$ container $c1$ should be in pile $p2$
    - $\neg in(c2, p3) \Longrightarrow$ container $c2$ should not be in pile $p3$
- Thus, $S_g = \{s \in S | s\ satisfies\ g\}$
  - positive atoms in g are also in s
  - negated atoms in g are not in s



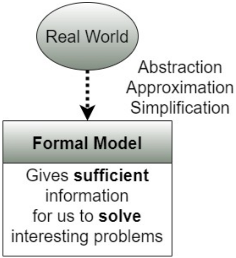
More expressive than positive goals

Still not as expressive as the STS:
"arbitrary set of states"

Many classical planners use one of these two alternatives (atoms/literals); some are more expressive

Abstraction

- We have abstracted the real world!
  - Motion is really continuous in 3D space
    - Uncountably infinite number of positions for a crane
- But for the purpose of planning:
  - We model a finite number of interesting positions
    - On a specific robot
    - In a specific pile
    - Held by a specific crane



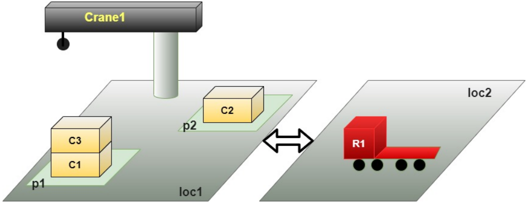
Actions with structure

- Make sense for action to have an internal structure!
  - $\gamma(s_{291823}, a_{120938}) = \emptyset \Longrightarrow$ "action move(A,p1,p3) requires a state where on(A,p1)"
  - $\gamma(s_{291823}, a_{120938}) = \{s_{12578942}\} \Longrightarrow$ "action move(A,p1,p3) makes on(A,p3) true, and..."

Operators

In the classical representation: Do not define actions directly

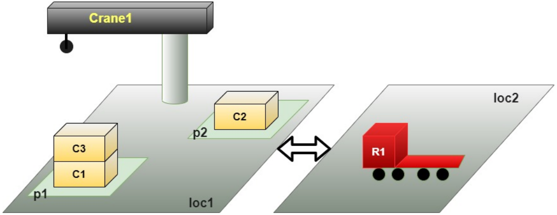
- Define a set O of operators
- Each operator is parameterized, defines many actions
  - *;; crane k at location l takes container c off container d in pile p" $\Longrightarrow$ take(k,l,c,d,p)*
- Has a precondition
  - precond(o): set of literals that must hold before execution
  - precond(take) = {belong(k,l), 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), ¬ in(c,d), top(d,p) }



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) preconditions and 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,c1), top(c1,p1) }

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



Untype Actions and Applicability

If every ground instantiation of an operator is an action...

- ... then it is this:
  - take(c3, crane1, r1, crane2, r2)
    ;; Container c3 at location crane1 takes r1 off crane2 in pile r2
- But when will this action be applicable?
  - take(k,l,c,d,p) *;; crane k at location l takes container c off container d in pile p*
    precond: {belong(k,l), empty(k), attached(p,l), top(c,p), on(c,d)}
  - take(c3,crane1,r1,crane2,r2)
    precond: {belong(c3,crane1), empty(c3), attached(r2,crane1), top(r1,r2), on(r1,crane2)}

For these preconditions to be true, something must already have gone wrong!

Untype Actions and Applicability

More common solution: Separate type predicates

- Ordinary predicates that happen to represent types:
  - crane(x), location(x), container(x), pile(x)
- Used as part of preconditions:
  - take(k,l,c,d,p) *;; crane k at location l takes container c off container d in pile p*
    precond: {crane(k), location(l), container(c), container(d), pile(p),
    belong(k,l), empty(k), attached(p,l), top(c,p), on(c,d) }
- DWR example was "optimized" somewhat
  - belong(k,l) is only true for crane+location, replaces two type predicates
- So...
  - take(c3,crane1,r1,crane2,r2) is an action
  - Its preconditions can never be satisfied in reachable states!
  - Type predicates are fixed, rigid, never modified
    $\Longrightarrow$ such actions can be filtered out before planning even starts

Useful properties

- If a is an operator or action...
  - precond+(a) = {atoms that appear positively in a's preconditions}
  - precond-(a) = {atoms that appear negated in a's preconditions}
  - effects+(a) = {atoms that appear positively in a's effects}
  - effects-(a) = {atoms that appear negated in a's effects}
- Example
  - take(k,l,c,d,p):

    ;; crane k at location l takes container c off container d in pile p

    precond(a) : belong(k,l), empty(k), attached(p,l), top(c,p), on(c,d)

    effect(a) : holding(k,c), ¬ empty(k), ¬ in(c,p), ¬ top(c,p), ¬ on(c,d), top(d,p)

  - effects+(a) = {holding(k,c), top(d,p)}
  - effects-(a) = {empty(k), in(c,p), top(c,p), on(c,d) }

Negation disappears!

Applicable (Executable) Actions

- An action a is applicable in a state s ...
  - ... if precond+(a) $\subseteq$ s and precond-(a) $\cap s = \emptyset$
- Example
  - take(crane1,loc1,c3,c1,p1):
    ;; crane1 at loc1 takes c3 off c1 in pile p1
    precond(a) : { belong(crane1,loc1), empty(crane1), attached(p1,loc1), top(c3,p1), on(c3,c1) }
    effect(a) : { 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) }

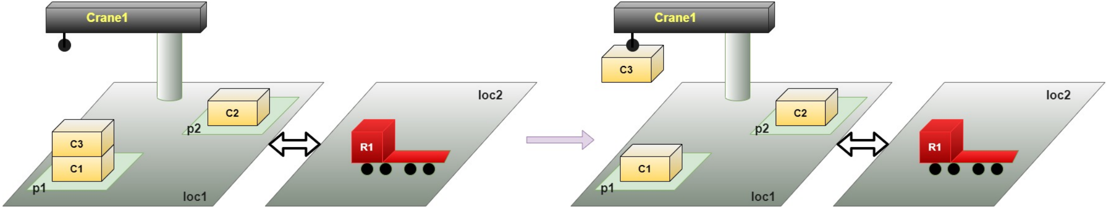
Result of Performing an Action

- Applying an action will add positive effects, delete negative effects
  - If a is applicable in s, then the new state is $(s \setminus$ effects-(a)$) \cup$ effects+(a)
- Example
  - take(crane1,loc1,c3,c1,p1):

    ;; crane1 at loc1 takes c3 off c1 in pile p1

    precond(a) : { belong(crane1,loc1), empty(crane1), attached(p1,loc1), top(c3,p1), on(c3,c1)}

    effect(a) : { holding(crane1,c3), ¬ empty(crane1), ¬ in(c3,p1), ¬ top(c3,p1), ¬ on(c3,c1), top(c1,p1)}



Defining γ

Positive preconditions missing from state

Negated preconditions present in state

$$\gamma(s,a) = \begin{cases} \emptyset & \text{if } \mathit{precond}+(a) \not\subseteq s \text{ or } \mathit{precond}-(a) \cap s \neq \emptyset \\ (s \setminus \mathit{effect}-(a)) \cup \mathit{effect}+(a) & \mathit{otherwise} \end{cases}$$

From the classical representation language, we know how to define $\Sigma = (S, A, \gamma)$, and a problem (Σ, s_0, S_g).

Modeling: What is a Precondition?

These are *physical* requirements for taking a container!

- Usual assumption in domain-independent planning:
  - Preconditions should have to do with *executability*, not *suitability*
    - Weakest constraints under which the action can be executed

take(crane1,loc1,c3,c1,p1):
;; crane1 at loc1 takes c3 off c1 in pile p1
precond(a) : { belong(crane1,loc1), empty(crane1), attached(p1,loc1), top(c3,p1), on(c3,c1) }
effect(a) : { holding(crane1,c3), ¬ empty(crane1), ¬ in(c3,p1), ¬ top(c3,p1), ¬ on(c3,c1), top(c1,p1)}

  - The *planner* chooses which actions are *suitable*, using heuristics (etc.)
  - Add explicit "suitability preconditions" $\Longrightarrow$ *domain-configurable planning*
    - "Only pick up a container if there is a truck on which the crane can put it"
    - "Only pick up a container if it needs to be moved according to the goal"

Domain-Independent Planning

High level problem description

Objects, Predicates, Operators,
Initial state, Goal

$\Downarrow$

Domain independent classical planner

Written for generic planning problems
Difficult to create (but done *once*)
Improvements $\Longrightarrow$ all domains benefit

$\Downarrow$

Solution (Plan)

Domain vs Instance

Domain description: "The world in general"	Instance description: "Our current problem"
Predicates Operators	Objects Initial state Goal
⇓	⇓

Domain-independent Planner

Domain-Independent Planning

- To solve problems in other domains:
  - Keep the planning algorithm
  - Write a new high-level description of the problem domain