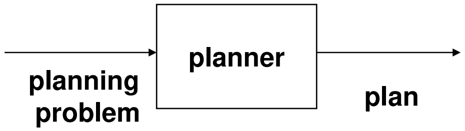
8 Strong Method Problem Solving

Note: the material for Section 8.4 is significantly enhanced

What is planning?

- A *planner* is a system that finds a sequence of actions to accomplish a specific task
- A planner <u>synthesizes</u> a plan



What is planning? (cont'd)

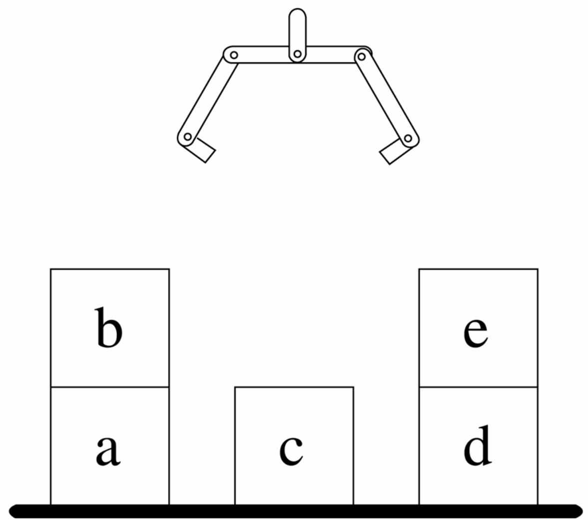
- The main components of a *planning problem* are:
  - a description of the starting situation (*the initial state*),
  - a description of the desired situation (*the goal state*),
  - the actions available to the executing agent (*operator library*, a.k.a. *domain theory*).
- Formally, a (classical) planning problem is a triple: <I, G, D>

where, I is the initial state,
G is the goal state, and
D is the domain theory.

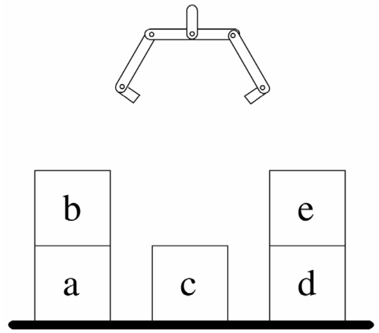
Characteristics of classical planners

- They need a mechanism to reason about actions and the changes they inflict on the world
- Important assumptions:
  - the agent is the only source of change in the world, otherwise the environment is static
  - all the actions are deterministic
  - the agent is omniscient: knows everything it needs to know about start state and effects of actions
  - the goals are categorical, the plan is considered successful iff all the goals are achieved

The blocks world



Represent this world using predicates



ontable(a)
ontable(c)
ontable(d)
on(b,a)
on(e,d)
clear(b)
clear(c)
clear(e)
gripping()

Declarative (or procedural) rules

If a block is clear, then there are no blocks on top of it (declarative)

OR

To make sure that a block is clear, make sure to remove all the blocks on top of it (procedural)

1. $(\forall X) \quad (\text{clear}(X) \leftrightarrow \neg (\exists Y) (\text{on}(Y, X)))$

Another example:
In order to fly to San Francisco, you need to have a ticket
vs.
In order to fly to San Francisco, make sure you that you have (bought) a ticket

Declarative (or procedural) rules

If a block is on the table, it is not on another block.

2. $(\forall Y)(\forall X) \quad \neg\, \text{on}(Y, X) \leftrightarrow \text{ontable}(Y)$

If the gripper is holding nothing, it is not holding anything

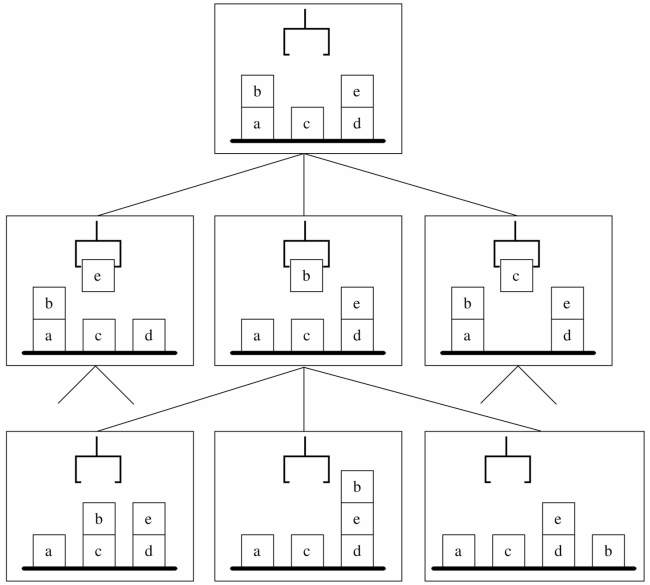
3. $(\forall Y) \quad \text{gripping}() \leftrightarrow \neg\, \text{gripping}(Y)$

The robot arm can perform these tasks

- **pickup (W)**: pick up block W from its current location on the table and hold it
- **putdown (W)**: place block W on the table
- **stack (U, V)**: place block U on top of block V
- **unstack (U, V)**: remove block U from the top of block V and hold it

All assume that the robot arm can precisely reach the block.

Portion of the search space or the blocks world example



The STRIPS representation

Special purpose representation.

An operator is defined in terms of its:

name,
parameters,
preconditions, and
results.

A planner is a special purpose algorithm, i.e., it's not a general purpose logic theorem prover.(We'll discuss this later.)

Four operators for the blocks world

pickup(X)	P: gripping() $\wedge$ clear(X) $\wedge$ ontable(X) A: gripping(X) D: ontable(X) $\wedge$ gripping()
putdown(X)	P: gripping(X) A: ontable(X) $\wedge$ gripping() $\wedge$ clear(X) D: gripping(X)
stack(X,Y)	P: gripping(X) $\wedge$ clear(Y) A: on(X,Y) $\wedge$ gripping() $\wedge$ clear(X) D: gripping(X) $\wedge$ clear(Y)
unstack(X,Y)	P: gripping() $\wedge$ clear(X) $\wedge$ on(X,Y) A: gripping(X) $\wedge$ clear(Y) D: on(X,Y) $\wedge$ gripping()

Notice the simplification

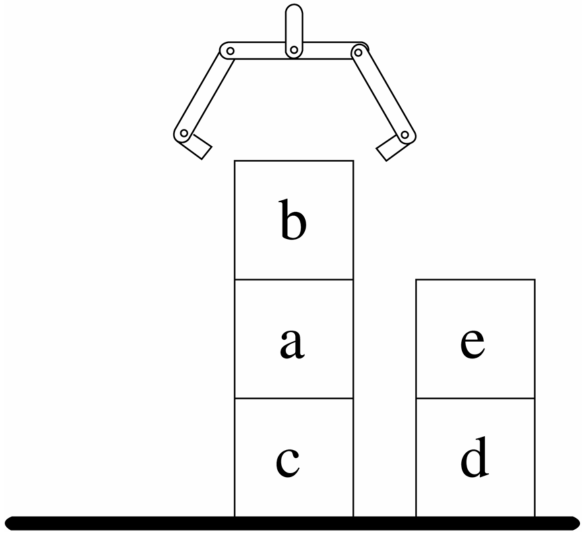
Preconditions, ***add lists***, and ***delete lists*** are all conjunctions. We don't have the full power of predicate logic.

The same applies to *goals*. Goals are conjunctions of predicates.

A detail:

Why do we have two operators for picking up (pickup and unstack), and two for putting down (putdown and stack)?

A goal state for the blocks world



A state space algorithm for STRIPS operators

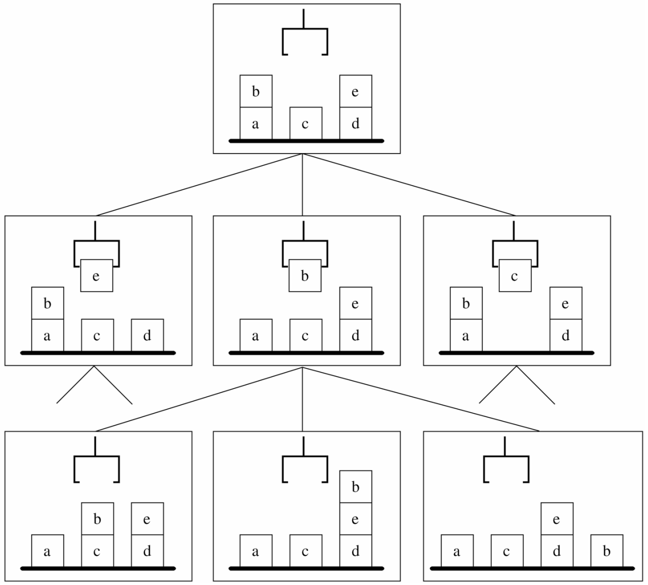
Search the space of situations (or states). This means each node in the search tree is a state.

The root of the tree is the start state.

Operators are the means of transition from each node to its children.

The goal test involves seeing if the set of goals is a subset of the current situation.

Now, the following graph makes much more sense



Problems in representation

Frame problem: **List everything that does not change. It no more is a significant problem because what is not listed as changing (via the add and delete lists) is assumed to be not changing.**

Qualification problem: **Can we list every precondition for an action? For instance, in order for PICKUP to work, the block should not be glued to the table, it should not be nailed to the table, …**

It still is a problem. A partial solution is to prioritize preconditions, i.e., separate out the preconditions that are worth achieving.

Problems in representation (cont'd)

Ramification problem: **Can we list every result of an action? For instance, if a block is picked up its shadow changes location, the weight on the table decreases, ...**

It still is a problem. A partial solution is to code rules so that inferences can be made. For instance, allow rules to calculate where the shadow would be, given the positions of the light source and the object. When the position of the object changes, its shadow changes too.

The gripper domain

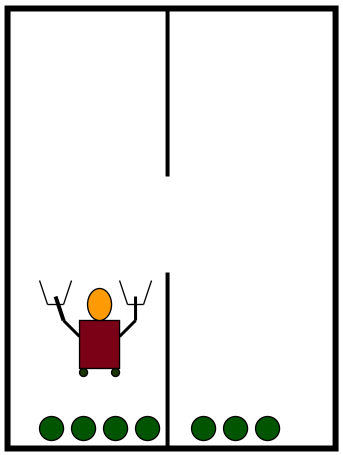
The agent is a robot with two grippers (left and right)

There are two rooms (rooma and roomb)

There are a number of balls in each room

Operators:

- PICK
- DROP
- MOVE



A "deterministic" plan

Pick ball1 rooma right

Move rooma roomb

Drop ball1 roomb right

Remember: the plans are generated "offline," no observability, nothing can go wrong

The gripper domain is interesting because parallelism is possible: can pick with both grippers at the same time

How to define a planning problem

- Create a *domain file:* contains the domain behavior, simply the operators
- Create a *problem file:* contains the initial state and the goal

The *domain definition* for the gripper domain

```
(define (domain gripper-strips)
  (:predicates   (room ?r)          (ball ?b)
                 (gripper ?g)       (at-robby ?r)
                 (at ?b ?r)         (free ?g)
                 (carry ?o ?g))
```

name of the domain

name of the action

"?" indicates a variable

combined add and delete lists

```
 (:action move
        :parameters (?from ?to)
        :precondition (and (room ?from) (room ?to)
                        (at-robby ?from))
        :effect (and (at-robby ?to)
                 (not (at-robby ?from))))
```


The *domain definition* for the gripper domain (cont'd)

```
(:action pick
        :parameters (?obj ?room ?gripper)
        :precondition (and (ball ?obj) (room ?room)
          (gripper ?gripper) (at ?obj ?room)
          (at-robby ?room) (free ?gripper))
        :effect (and (carry ?obj ?gripper)
          (not (at ?obj ?room)) (not (free ?gripper))))
```


The *domain definition* for the gripper domain (cont'd)

```
(:action drop
        :parameters (?obj ?room ?gripper)
        :precondition (and (ball ?obj) (room ?room)
          (gripper ?gripper) (at-robby ?room)
          (carrying ?obj ?gripper))
        :effect (and (at ?obj ?room) (free ?gripper)
           (not (carry ?obj ?gripper)))))
```

An example *problem definition* for the gripper domain

```
(define (problem strips-gripper2)
  (:domain gripper-strips)

  (:objects rooma roomb ball1 ball2 left right)

  (:init (room rooma)         (room roomb)
         (ball ball1)         (ball ball2)
         (gripper left)       (gripper right)

         (at-robby rooma)
         (free left)          (free right)
         (at ball1 rooma)     (at ball2 rooma) )

  (:goal (at ball1 roomb)))
```


Running VHPOP

Once the domain and problem definitions are in files gripper-domain.pddl and gripper-2.pddl respectively, the following command runs Vhpop:

```
vhpop gripper-domain.pddl gripper-2.pddl
```

The output will be:

```
;strips-gripper2
1:(pick ball1 rooma right)
2:(move rooma roomb)
3:(drop ball1 roomb right)
Time: 0 msec.
```

"pddl" is the planning domain definition language.

Why is planning a hard problem?

It is due to the large branching factor and the overwhelming number of possibilities.

There is usually no way to separate out the relevant operators. Take the previous example, and imagine that there are 100 balls, just two rooms, and two grippers. Again, the goal is to take 1 ball to the other room.

How many PICK operators are possible in the initial situation?

```
pick
    :parameters (?obj ?room ?gripper)
```

That is only one part of the branching factor, the robot could also move without picking up anything.

Why is planning a hard problem? (cont'd)

Also, goal interactions is a major problem. In planning, goal-directed search seems to make much more sense, but unfortunately cannot address the exponential explosion. This time, the branching factor increases due to the many ways of resolving the interactions.

When subgoals are compatible, i.e., they do not interact, they are said to be *linear* (or *independent*, or *serializable*).

Life is easier for a planner when the subgoals are independent because then divide-and-conquer works.

How to deal with the exponential explosion?

Use goal-directed algorithms

Use domain-independent heuristics

Use domain-dependent heuristics (need a language to specify them)

The “monkey and bananas” problem

The “monkey and bananas” problem (cont’d)

The problem statement: A monkey is in a laboratory room containing a box, a knife and a bunch of bananas. The bananas are hanging from the ceiling out of the reach of the monkey. How can the monkey obtain the bananas?

?

VHPOP coding

```
(define (domain monkey-domain)
  (:requirements :equality)
  (:constants monkey box knife glass water
    waterfountain)
  (:predicates (on-floor) (at ?x ?y) (onbox ?x)
    (hasknife) (hasbananas) (hasglass) (haswater)
    (location ?x)

  (:action go-to
    :parameters (?x ?y)
    :precondition (and (not = ?y ?x)) (on-floor)
      (at monkey ?y)
    :effect (and (at monkey ?x) (not (at monkey ?y))))
```

VHPOP coding (cont'd)

```
(:action climb
  :parameters (?x)
  :precondition (and (at box ?x) (at monkey ?x))
  :effect (and (onbox ?x) (not (on-floor))))

(:action push-box
  :parameters (?x ?y)
  :precondition (and (not (= ?y ?x)) (at box ?y)
   (at monkey ?y) (on-floor))
  :effect (and (at monkey ?x) (not (at monkey ?y))
   (at box ?x) (not (at box ?y))))
```


VHPOP coding (cont'd)

```
(:action getknife
  :parameters (?y)
  :precondition (and (at knife ?y) (at monkey ?y))
  :effect (and (hasknife) (not (at knife ?y))))

(:action grabbananas
  :parameters (?y)
  :precondition (and (hasknife) (at bananas ?y)
   (onbox ?y) )
  :effect (hasbananas))
```


VHPOP coding (cont'd)

```
(:action pickglass
  :parameters (?y)
  :precondition (and (at glass ?y) (at monkey ?y))
  :effect (and (hasglass) (not (at glass ?y))))

(:action getwater
  :parameters (?y)
  :precondition (and (hasglass) (at waterfountain ?y)
   (ay monkey ?y) (onbox ?y))
  :effect (haswater))
```


Problem 1: monkey-test1.pddl

```
(define (problem monkey-test1)
  (:domain monkey-domain)
  (:objects p1 p2 p3 p4)
  (:init (location p1) (location p2)
         (location p3) (location p4)
         (at monkey p1) (on-floor)
         (at box p2) (at bananas p3)
         (at knife p4))
  (:goal (hasbananas)))
```

go-to p4 p1
get-knife p4
go-to p2 p4
push-box p3 p2
climb p3
grab-bananas p3 **time = 30 msec.**

Problem 2: monkey-test2.pddl

```
(define (problem monkey-test2)
  (:domain monkey-domain)
  (:objects p1 p2 p3 p4 p6)
  (:init (location p1) (location p2)
         (location p3) (location p4) (location p6)
         (at monkey p1) (on-floor)
         (at box p2) (at bananas p3) (at knife p4)
         (at waterfountain p3) (at glass p6))
  (:goal (and (hasbananas) (haswater))))
```

go-to p4 p1
get-knife p4
go-to p6 p4
pickglass p6
go-to p2 p6
push-box p3 p2
climb p3
getwater p3
grab-bananas p3

time = 70 msec.

The “monkey and bananas” problem
(cont’d) (Russell & Norvig, 2003)

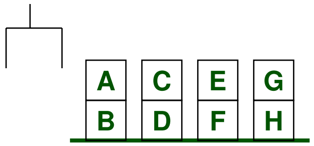
Suppose that the monkey wants to fool the scientists, who are off to tea, by grabbing the bananas, but leaving the box in its original place. Can this goal be solved by a STRIPS-style system?

A sampler of planning algorithms

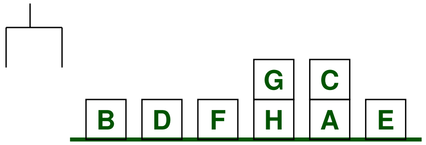
- **Forward chaining**
  - **Work in a state space**
  - **Start with the initial state, try to reach the goal state using forward progression**
- **Backward chaining**
  - **Work in a state space**
  - **Start with the goal state, try to reach the initial state using backward regression**
- **Partial order planning**
  - **Work in a plan space**
  - **Start with an empty plan, work from the goal to reach a complete plan**

Forward chaining

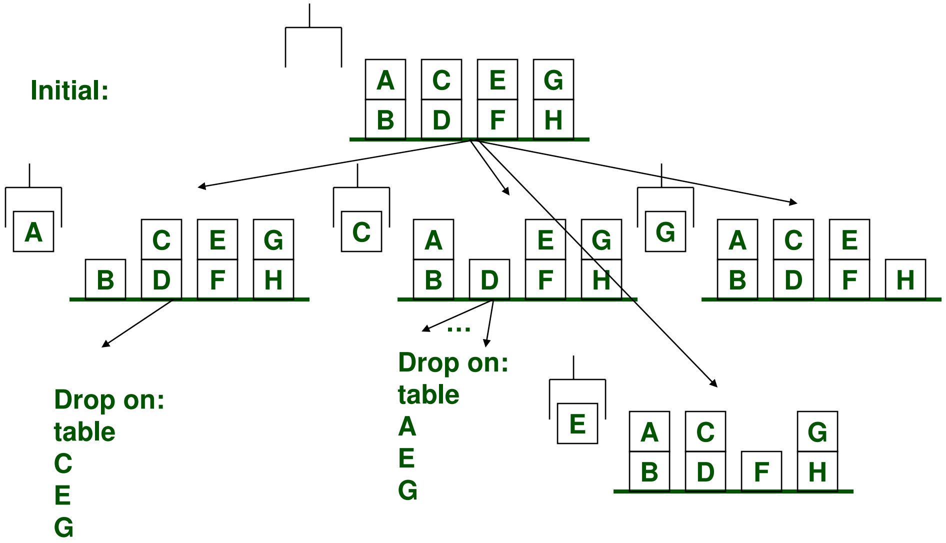
Initial:



Goal :



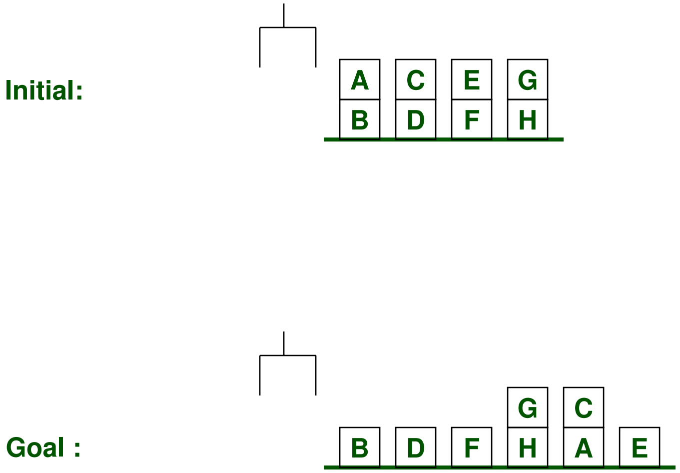
1st and 2nd levels of search



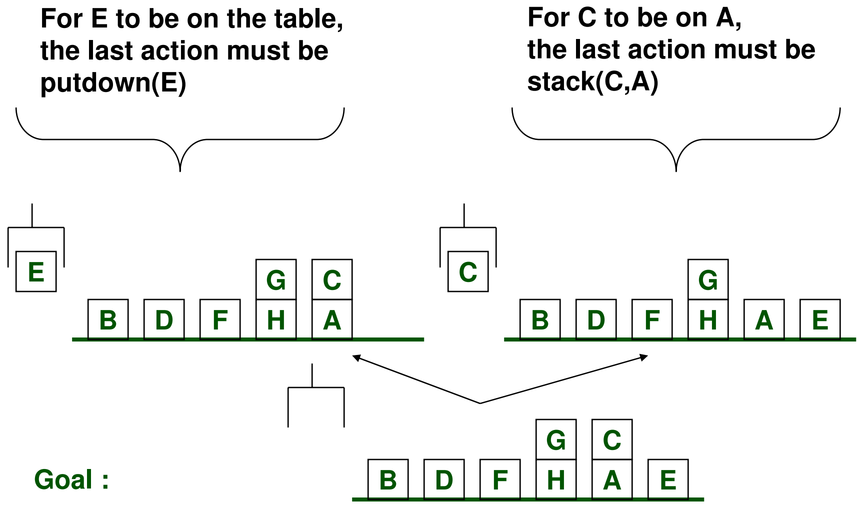
Results

- A plan is:
  - unstack (A, B)
  - putdown (A)
  - unstack (C, D)
  - stack (C, A)
  - unstack (E, F)
  - putdown (F)
- Notice that the final locations of D, F, G, and H need not be specified
- Also notice that D, F, G, and H will never need to be moved. But there are states in the search space which are a result of moving these. Working backwards from the goal might help.

Backward chaining

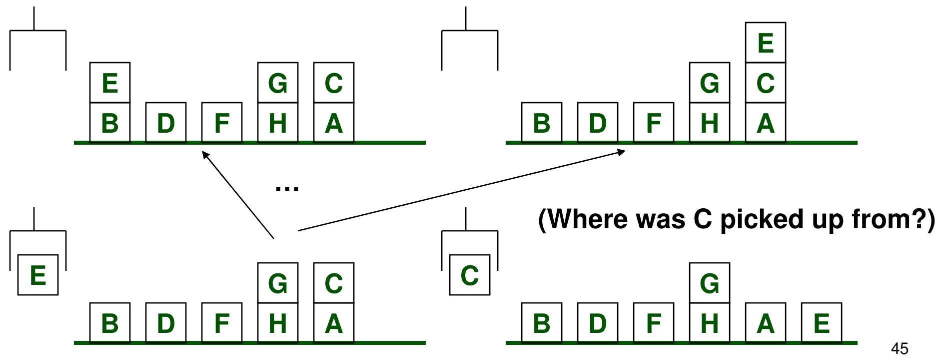


1st level of search



2nd level of search

Where was E picked up from?



Results

- The same plan can be found
  - unstack (A, B)
  - putdown (A)
  - unstack (C, D)
  - stack (C, A)
  - unstack (E, F)
  - putdown (F)
- Now, the final locations of D, F, G, and H need to be specified
- Notice that D, F, G, and H will never need to be moved. But observe that from the second level on the branching factor is still high

Partial-order planning (POP)

- Notice that the resulting plan has two parallelizable threads:

unstack (A,B)
putdown (A)
unstack (C,D)
stack (C,A)

&

unstack (E, F)
putdown (F)

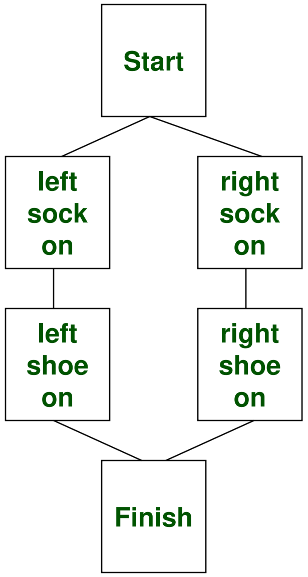
- These steps can be interleaved in 3 different ways:

unstack (E, F)	unstack (A,B)	unstack (A,B)
putdown (F)	putdown (A)	putdown (A)
unstack (A,B)	unstack (E, F)	unstack (C,D)
putdown (A)	putdown (F)	stack (C,A)
unstack (C,D)	unstack (C,D)	unstack (E, F)
stack (C,A)	stack (C,A)	putdown (F)

Partial-order planning (cont'd)

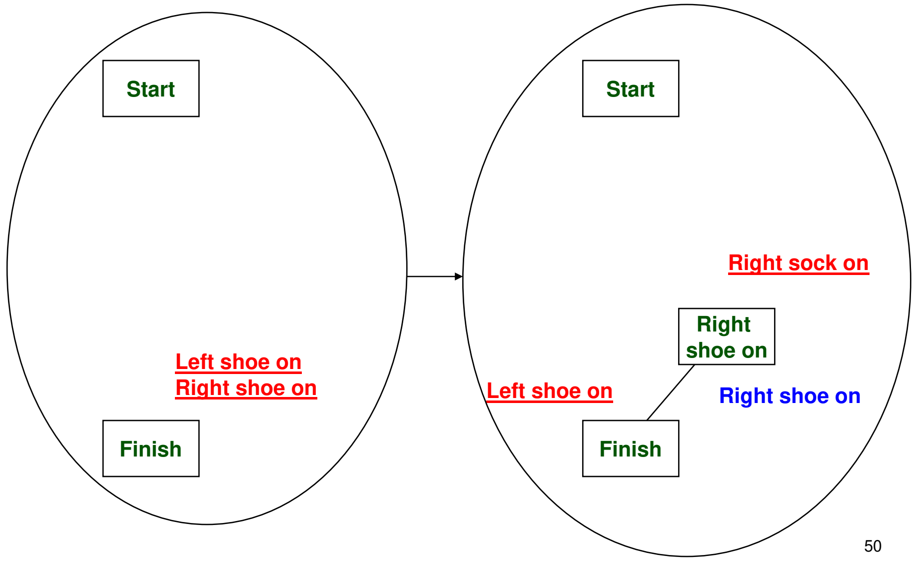
- Idea: Do not order steps unless it is necessary
- Then a partially ordered plan represents several totally ordered plans
- That decreases the search space
- But still the planning problem is not solved, good heuristics are crucial

Partial-order planning (cont'd)

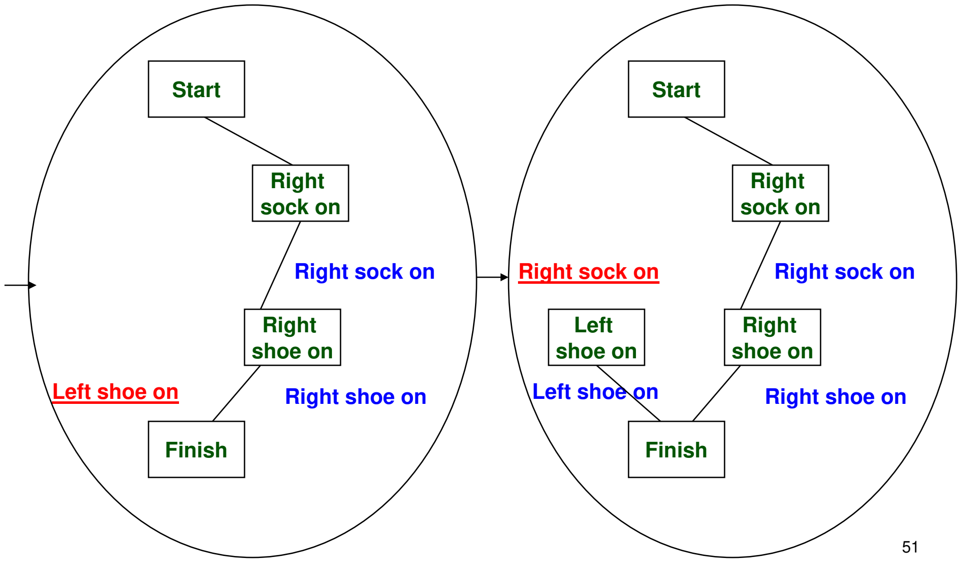


Start	Start	Start	Start	Start	Start
Left sock on	Right sock on	Left sock on	Right sock on	Left sock on	Right sock on
Left shoe on	Right shoe on	Right sock on	Left sock on	Right sock on	Left sock on
Right sock on	Left sock on	Left shoe on	Right shoe on	Right shoe on	Left shoe on
Right shoe on	Left shoe on	Right shoe on	Left shoe on	Left shoe on	Right shoe on
Finish	Finish	Finish	Finish	Finish	Finish

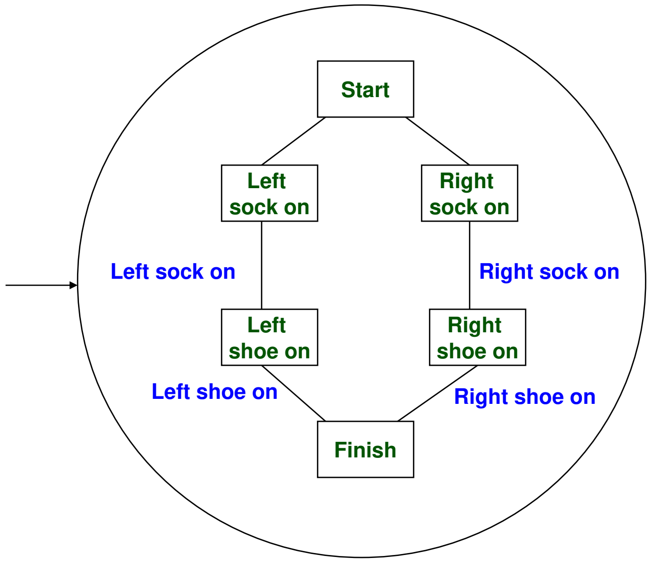
POP plan generation



POP plan generation (cont'd)



POP plan generation (cont'd)



DONE!

Comments on partial order planning

- The previous plan was generated in a straightforward manner but usually extensive search is needed
- In the previous example there was always just one plan in the search space, normally there will be many (see the GRIPPER results)
- There is no explicit notion of a state

Sample runs with VHPOP

- Ran increasingly larger gripper problems on wopr
- S+OC is the older heuristic: the estimated number of steps to complete the plan is number of steps + number of open conditions
- ADD uses a *plan graph* to estimate the "distance" to a complete plan
- Both heuristics are domain independent

Sample runs with VHPOP (cont'd)

In the examples/ directory

../vhpop –f static –h S+OC gripper-domain.pddl gripper-2.pddl

../vhpop –f static –h ADD gripper-domain.pddl gripper-2.pddl

Run times in milliseconds

Gripper Problem	Number of Steps	S+OC heuristic	ADD heuristic
2	3	2	13
4	9	193	109
6	15	79734	562
8	21	> 10 min	1937
10	27	---	4691
12	33	---	17250
20	59	---	326718

Could we have done this in logic?

Yes.

A block is clear if there are no blocks on top of it

1. $(\forall X)$ $(\ clear(X) \leftrightarrow \neg (\exists Y)\ (\ on(Y, X)\ ))$

If a block is on the table, it is not on another block.

2. $(\forall Y)(\forall X)$ $\neg\ on(Y, X) \leftrightarrow ontable(Y)$

If the gripper is holding nothing, it is not holding anything

3. $(\forall Y)$ $gripping() \leftrightarrow \neg\ gripping(Y)$

Rules for operations on the states

4. $(\forall X)$ $pickup(X) \rightarrow$
$(gripping(X) \leftarrow (gripping() \wedge clear(X) \wedge ontable(X)))$

5. $(\forall X)$ $putdown(X) \rightarrow$
$(gripping() \wedge ontable(X) \wedge clear(X) \leftarrow (gripping(X)))$

6. $(\forall X)$ $stack(X,Y) \rightarrow$
$((on\ (X,Y) \wedge gripping() \wedge clear(X)) \leftarrow$
$(clear(Y) \wedge gripping(X))\ )$

7. $(\forall X)$ $unstack(X,Y) \rightarrow$
$((clear(Y) \wedge gripping(X)\ ) \leftarrow$
$(on(X,Y) \wedge clear(X) \wedge gripping())\ )$

The format of the rules

$$A \rightarrow (B \leftarrow C)$$

where, A is the operator

B is the "result" of the operation

C is the conditions that must be true in order for the operator to be executable

They tell what changes when the operator is executed (or applied)

But ...

We have no explicit notion of a "state" that changes over time as actions are performed.

Remember that predicate logic is "timeless", everything refers to the same time.

In order to work reasoning about actions into logic, we need a way to tell that changes are happening over discrete times (or situations.)

Situation calculus

We need to add an additional parameter which represents the state. We'll use s_0, …, s_n to represent states (a.k.a. situations).

Now we can say:

4. $(\forall X)$ $\quad$ $pickup(X, s_0) \rightarrow$
$(gripping(X, s_1) \leftarrow$
$(gripping(nil, s_0) \wedge clear(X, s_0) \wedge$
$ontable(X, s_0)))$

If the pickup action was attempted in state 0, with the conditions listed holding, then in state 1, gripping will be true for X.

Introduce "holds" and "result" and generalize over states

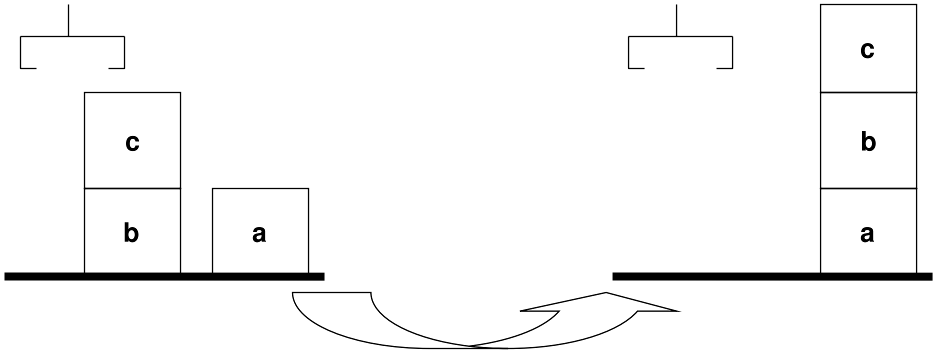
4. $(\forall X)\ (\forall s)$ $\quad$ $(holds(gripping(\ ), s) \wedge$
$holds(clear(X), s) \wedge$
$holds(ontable(X), s))$
$\rightarrow$
$(holds(gripping(X), result(pickup(X),s))$

Using rules like this we can logically prove what happens as several actions are applied consecutively.

Notice that gripping, clear, …, are now functions.

Is "result" a function or a predicate?

A small "plan"



(result(stack(c,b),
(result(pickup(c),
(result (stack(b, a),
(result(pickup(b),
(result(putdown(c),
(result(unstack(c,b),s0))))))

Our rules will still not work, because...

We are making an implicit (but big) assumption: we are assuming that if nothing tells us that p has changed, then p has not changed.

This is important because we want to reason about change, as well as no-change.

For instance, block a is still clear after we move block c around (except on top of block a).

Things are going to start to get messier because we now need ***frame axioms***.

A frame axiom

Tells what doesn't change when an action is performed.

For instance, if Y is "unstacked" from Z, nothing happens to X.

$(\forall X)\ (\forall Y)\ (\forall Z)\ (\forall s)$
$(holds\ (ontable(X),\ s)$
$\rightarrow$
$(holds(ontable(X),\ result(unstack(Y,\ Z),\ s)$

For our logic system to work, we'll have to define such an axiom for each action and for each predicate.

This is called the ***frame problem*** .

Comments on planning

- It is a synthesis task
- Classical planning is based on the assumptions of a deterministic and static environment
- Theorem proving and situation calculus are not widely used nowadays for planning (see below)
- Algorithms to solve planning problems include:
  - forward chaining: heuristic search in state space
  - Graphplan: mutual exclusion reasoning using plan graphs
  - Partial order planning (POP): goal directed search in plan space
  - Satifiability based planning: convert problem into logic

Comments on planning (cont'd)

- Non-classical planners include:
  - probabilistic planners
  - contingency planners (a.k.a. conditional planners)
  - decision-theoretic planners
  - temporal planners
  - resource based planners

Comments on planning (cont'd)

- In addition to plan generation algorithms we also need algorithms for
  - Carrying out the plan
  - Monitoring the execution
    (because the plan might not work as expected; or the world might change)
    (need to maintain the consistency between the world and the program's internal model of the world)
  - Recovering from plan failures
  - Acting on new opportunities that arise during execution
  - Learning from experience
    (save and generalize good plans)

Triangle table (execution monitoring and macro operators)

	1	2	3	4	5	6	7
1	gripping() clear(X) on(X,Y)	**unstack(X,Y)**					
2		gripping(X)	**putdown(X)**				
3	ontable(Y)	clear(Y)	gripping()	**pickup(Y)**			
4	clear(Z)			gripping(Y)	**stack(Y,Z)**		
5			clear(X) ontable(X)		gripping()	**pickup(X)**	
6					clear(Y)	gripping(X)	**stack(X,Y)**
7					on(Y,Z)		on(X,Y) clear(X) gripping()

Applications of planning

- **Robotics**
  - **Shakey, the robot at SRI was the initial motivator**
  - **However, several other techniques are used for path-planning etc.**
  - **Most robotic systems are *reactive***
- **Games**
  The story is a plan and a different one can be constructed for each game
- **Web applications**
  Formulating query plans, using web services
- **Crisis response**
  Oil spill, forest fire, emergency evacuation

Applications of planning (cont'd)

- **Space**
  Autonomous spacecraft, self-healing systems
- **Device control**
  Elevator control, control software for modular devices
- **Military planning**
- **And many others …**

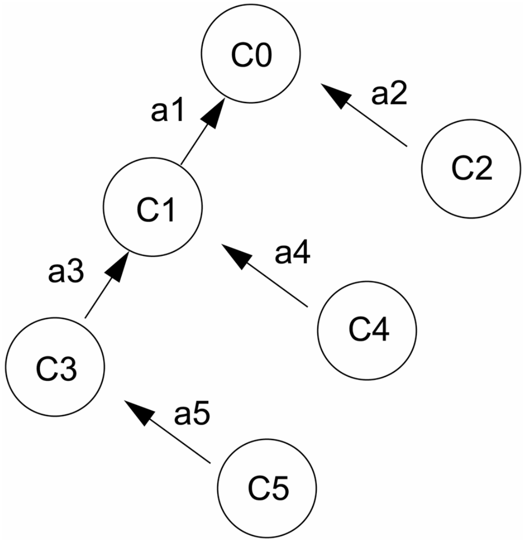
Model-based reactive configuration management (Williams and Nayak, 1996a)

Intelligent space probes that autonomously explore the solar system.

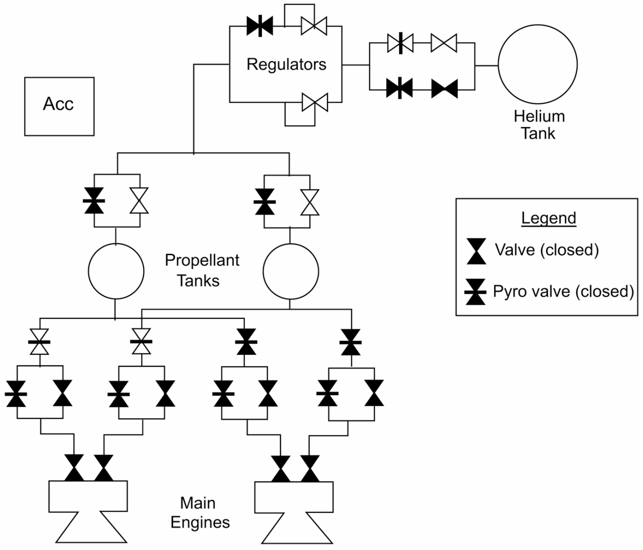
The spacecraft needs to:

- **radically reconfigure its control regime in response to failures,**
- **plan around these failures during its remaining flight.**

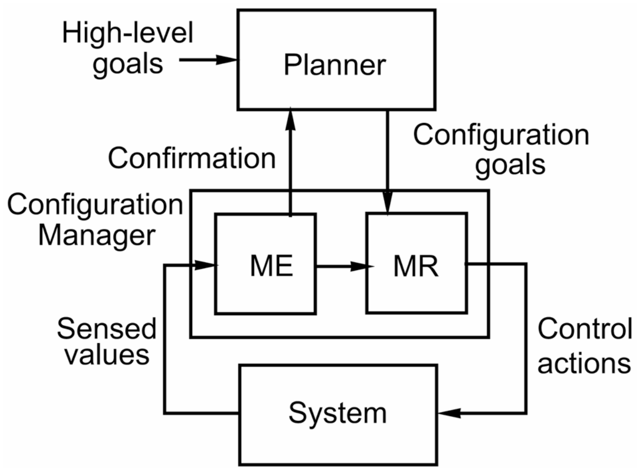
Teleo-reactive planning: combines feedback-based control and discrete actions (Klein et al., 2000)



A schematic of the simplified Livingstone propulsion system (Williams and Nayak ,1996)

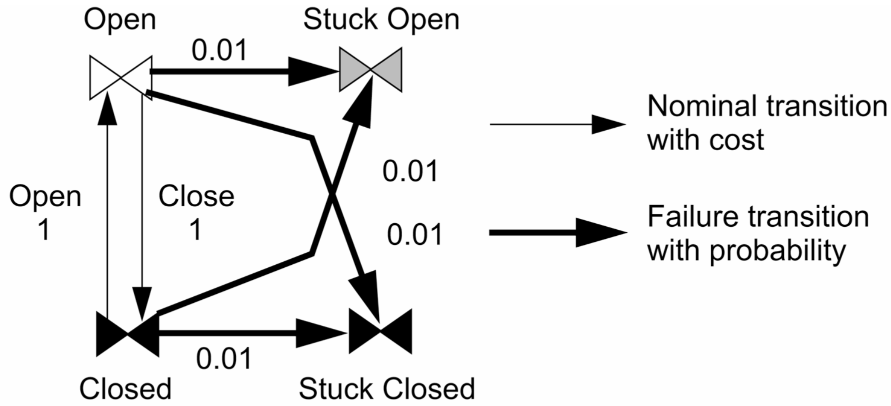


A model-based configuration management system (Williams and Nayak, 1996)

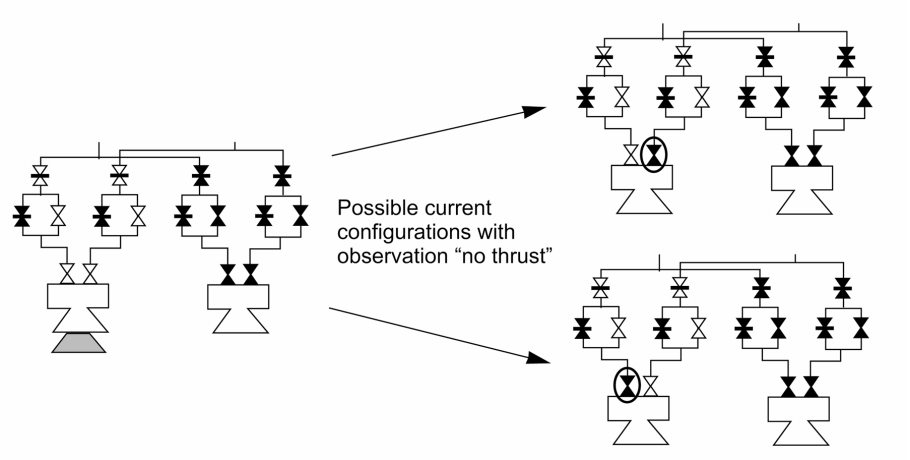


ME: mode estimation **MR: mode reconfiguration**

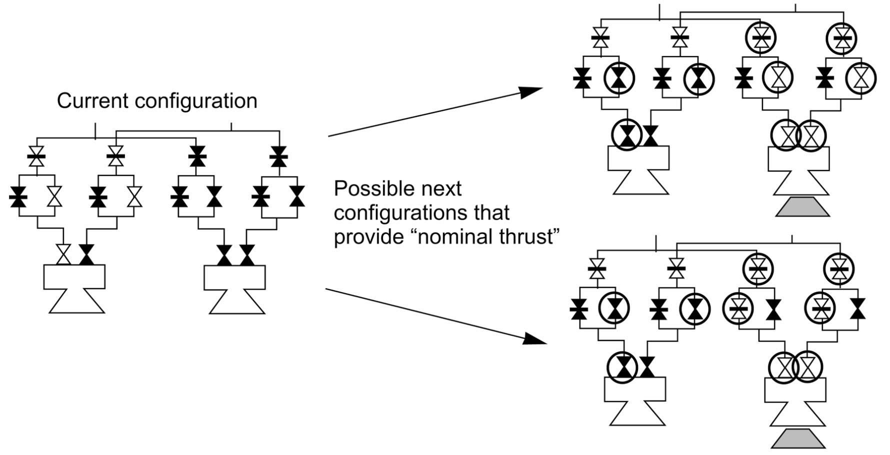
The transition system model of a valve (Williams and Nayak, 1996a)



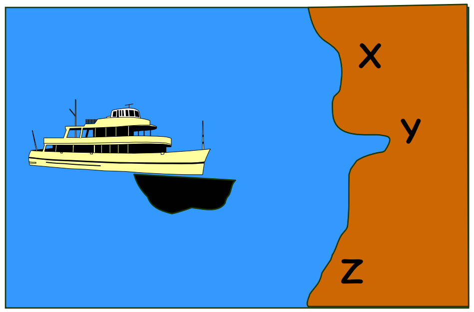
Mode estimation (Williams and Nayak, 1996a)



Mode reconfiguration (MR) (Williams and Nayak, 1996a)



Oil spill response planning

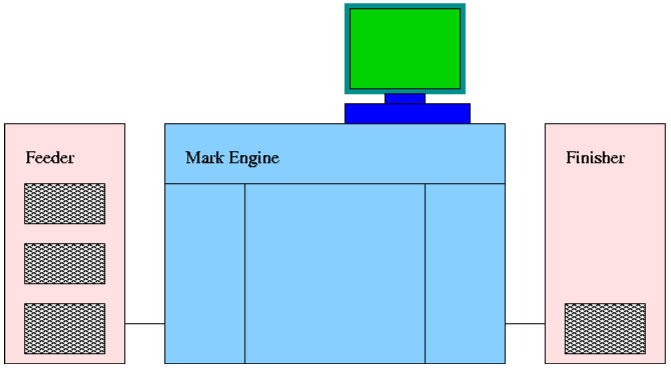


(Desimone & Agosto 1994)

Main goals: stabilize discharge, clean water, protect sensitive shore areas

The objective was to estimate the equipment required rather than to execute the plan

A modern photocopier



(From a paper by Fromherz et al. 2003)

Main goal: produce the documents as requested by the user

Rather than writing the control software, write a controller that produces and executes plans

The paper path