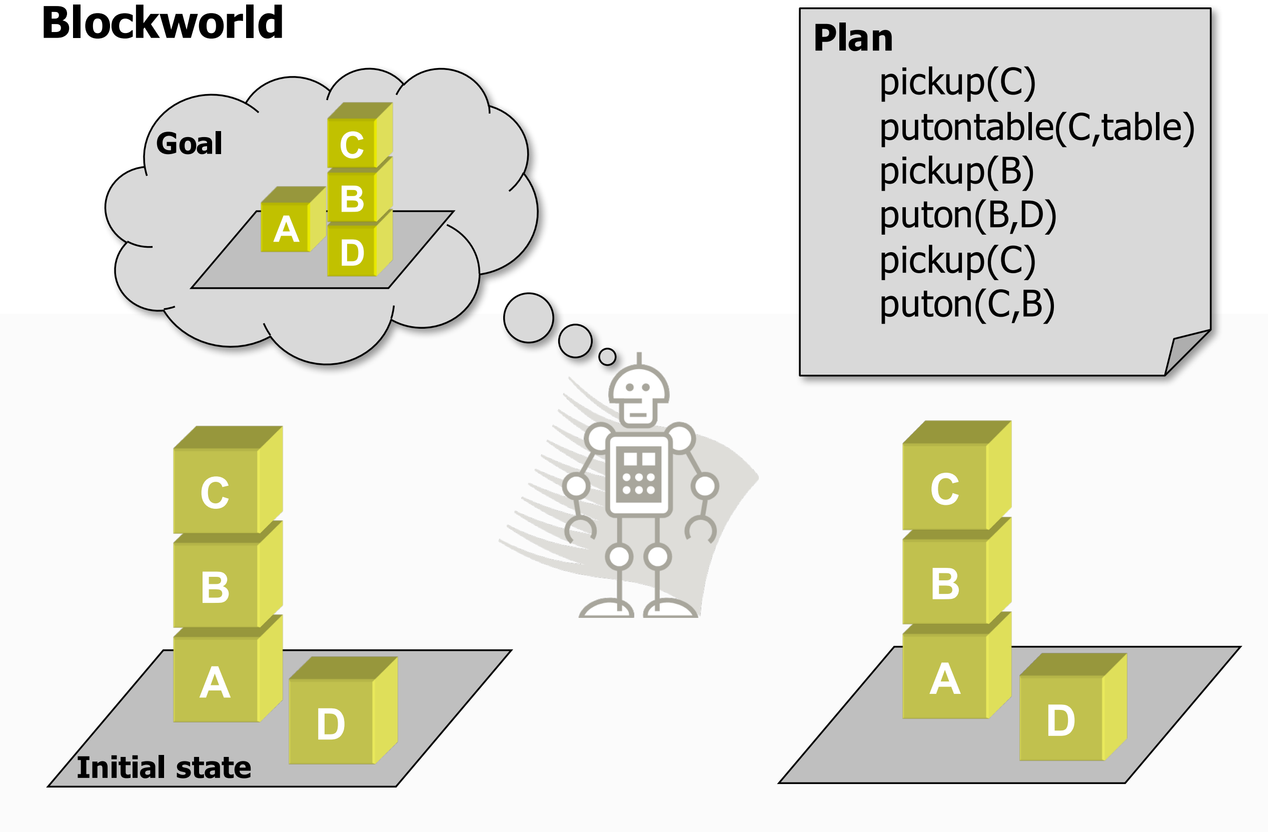
Automated Planning

Roman Barták
Charles University in Prague, Faculty of Mathematics and Physics

bartak@ktiml.mff.cuni.cz

What is planning?

Blockworld



Input:

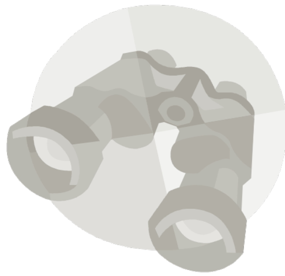
– initial (current) state of the world
– description of actions that can change the world
– desired state of the world

Output:

– a sequence of actions (a plan)

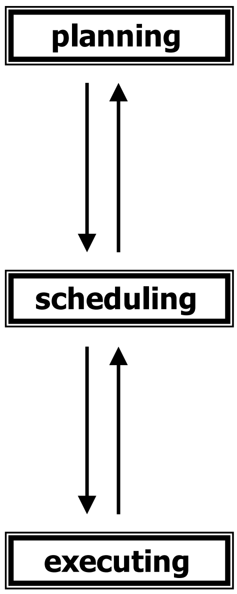
Properties:

– actions in the plan are unknown
– time and resources are not assumed



Planning

– deciding which actions are necessary to achieve the goals
– topic of artificial intelligence
– complexity is usually worse than NP-c (in general, undecidable)



Scheduling

– deciding how to process the actions using given restricted resources and time
– topic of operations research
– complexity is typically NP-c

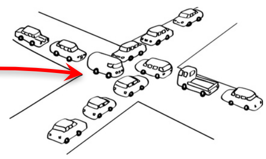
Deep Space 1

Launch: October 24, 1998

Target: Comet Borrelly

testing a payload of 12 advanced, high risk technologies

- **autonomous remote agent**
  - planning, execution, and monitoring spacecraft activities based on general commands from operators
  - three testing scenarios
    - 12 hours of low autonomy (execution and monitoring)
    - 6 days of high autonomy (operating camera, simulation of faults)
    - 2 days of high autonomy (keep direction)
      - » **beware of backtracking!**
      - » **beware of deadlock in plans!**



Tutorial outline

- **Problem Formalisation**
  - models and representations
- **State-space Planning**
  - forward and backward search
- **Plan-space Planning**
  - partial-order planning
- **Control Knowledge in Planning**
  - heuristics
  - control rules

Planning deals with **selection and organization of actions** that are changing world states.

System Σ modelling states and transitions:

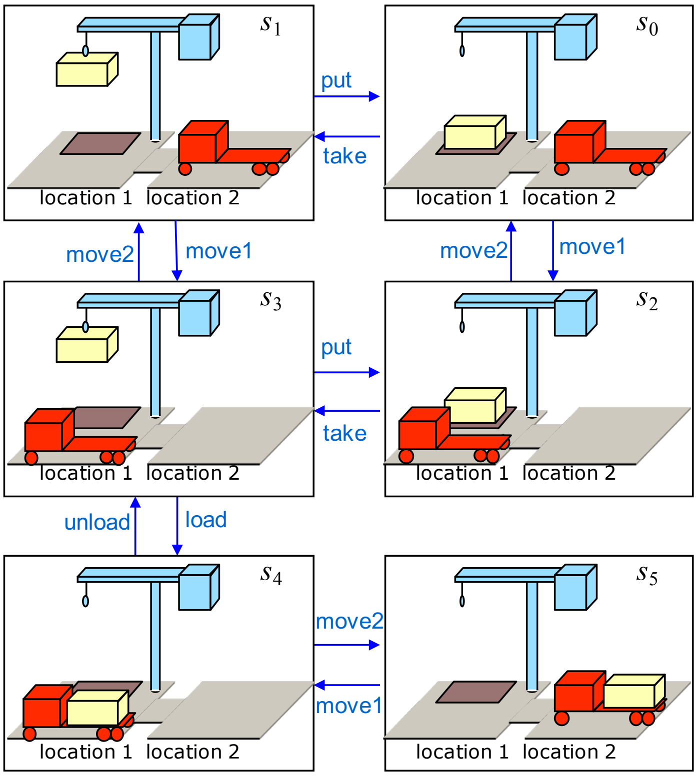
- **set of states S** (recursively enumerable)
- **set of actions A** (recursively enumerable)
  - actions are controlled by the planner!
  - no-op
- **set of events E** (recursively enumerable)
  - events are out of control of the planner!
  - neutral event ε
- **transition function** γ: S x A x E → 2^S
  - actions and events are sometimes applied separately
    γ: Sx(A ∪ E) → P(S)

A planning task is to find which actions are applied to world states to reach some goal from a given initial state.

What is a goal?

- **goal state** or a set of of goal states
- **satisfaction of some constraint** over a sequence of visited states
  - for example, some states must be excluded or some states must be visited
- **optimisation of some objective function** over a sequence of visited states (actions)
  - for example, maximal cost or a sum of costs

Example

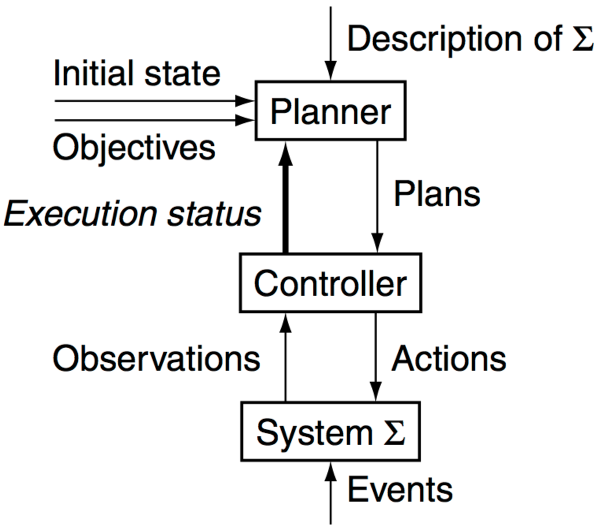


$\Sigma = (S, A, E, \gamma)$

- $S = \{s_0, \ldots, s_5\}$
- $E = \{\}$ resp. $\{\varepsilon\}$
- $A = \{$move1, move2, put, take, load, unload$\}$
- γ: see figure

- init: s_0
- goal: s_5

How does it work?



A **planner** generates plans

A **controller** takes care about plan execution

- for each state it selects an action to execute
- observations (sensor input) are translated to world state

Dynamic planning involves re-planning when the state is not as expected.

Some assumptions

- the system is **finite**
- the system is **fully observable**
  - We know the current state completely.
- the system is **deterministic**
  - $\forall s \in S\ \forall u \in (A \cup E)$: $|\gamma(s,u)| \leq 1$
- the system is **static**
  - There are no external events.
- the **goals** are **restricted**
  - The aim is to reach one of the goal states.
- the **plans** are **sequential**
  - A plan consists of a (linearly ordered) sequence of actions.
- **time** is **implicit**
  - Actions are instantaneous (no duration is assumed)).
- **planning** is done **offline**
  - State of the world does not change during planning.

Classical planning

We will work with a deterministic, static, finite, and fully observable state-transition system with restricted goals and implicit time $\Sigma = (S, A, \gamma)$.

Planning problem P = (Σ,s_0,g):

- s_0 is the **initial state**
- g describes the **goal states**

A solution to the planning problem P is a sequence of actions $\langle a_1, a_2, \ldots, a_k \rangle$ with a corresponding sequence of states $\langle s_0, s_1, \ldots, s_k \rangle$ such that $s_i = \gamma(s_{i-1}, a_i)$ and s_k satisfies g

☞ **Classical planning (STRIPS planning)** ☜

Simplification?

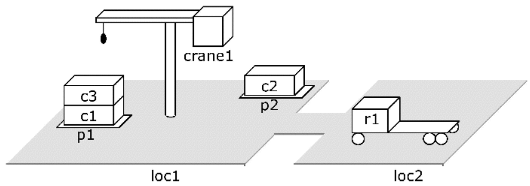
Planning in the restricted model reduces to “path finding” in the graph defined by states and state transitions.

Is it really so simple?

5 locations, 3 piles per location, 100 containers, 3 robots

↳ **10^{277} states**

This is 10^{190} times more than the largest estimate of the number of particles in the whole universe!



This tutorial

How to represent states and actions without enumerating the sets S and A?

– recall 10^{277} states with respect to the number of particles in the universe

How to efficiently solve planning problems?

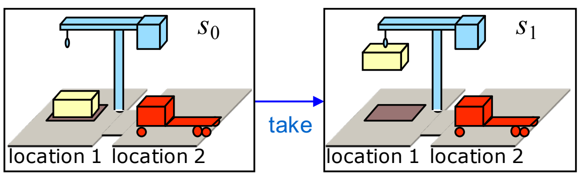
– How to find a path in a graph with 10^{277} nodes?

- **Problem Formalisation**
  - models and representations
- State-space Planning
  - Forward and backward search
- Plan-space Planning
  - Partial-order planning
- Control Knowledge in Planning
  - heuristics
  - control rules

Each **state** is described using a **set of propositions** that hold at that state.
example: {onground, at2}

Each **action** is a syntactic expression describing:

- which propositions must hold in a state so the action is applicable to that state
  example: take: {onground}
- which propositions are added and deleted from the state to make a new state
  example:
  take: {onground}⁻, {holding}⁺



Set representation: a planning domain

Let L= {p_1, …, p_n} be a finite set of propositional symbols (language).

A planning domain Σ over L is a triple (S,A,γ):

- S ⊆ 2^L, i.e. **state** s is a subset of L describing which propositions hold in that state
  - if **p** ∈ **s**, then **p** holds in **s**
  - if **p** ∉ **s**, then **p** does not hold in **s**
- **action** a ∈ A is a triple of subsets of L
  a = (precond(a),$effects^-$(a),$effects^+$(a))
  - $effects^-$(a) ∩ $effects^+$(a) = ∅
  - action **a** is applicable to state **s** iff precond(**a**) ⊆ **s**
- **transition function** γ:
  - γ(**s**,**a**) = (**s** – $effects^-$(**a**)) ∪ $effects^+$(**a**), if **a** is applicable to **s**

Set representation: a planning problem

Planning problem P is a triple (**Σ**,s_0,g):

- **Σ** = (S,A,γ) is a planning domain over L
- s_0 is an initial state, s_0 ∈ S
- g ⊆ L is a set of goal propositions
  - S_g = {s ∈ S | g ⊆ s} is a set of goal states

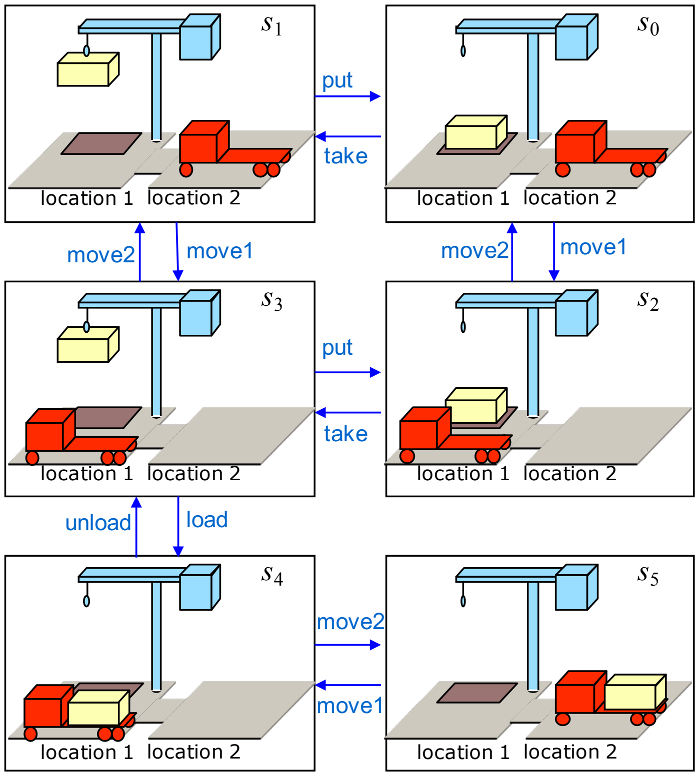
Plan π is a sequence of actions ⟨a_1,a_2,…,a_k⟩

- **the length of plan** π is k = | π |
- **a state obtained by the plan** π (a transitive closure of γ)
  - γ(s, π) = s, if k=0 (plan π is empty)
  - γ(s, π) = γ(γ(s,a_1), ⟨a_2,…,a_k⟩), if k>0 and a_1 is applicable to s
  - γ(s, π) = undefined, otherwise

Plan π is a **solution plan** for P iff g ⊆ γ(s_0, π).

- **redundant plan** contains a subsequence of actions that also solves P
- **minimal plan**: there is no shorter solution plan for P

Set representation: example



L = {onground, onrobot, holding, at1, at2}

s_0 = {onground, at2}

g = {onrobot}

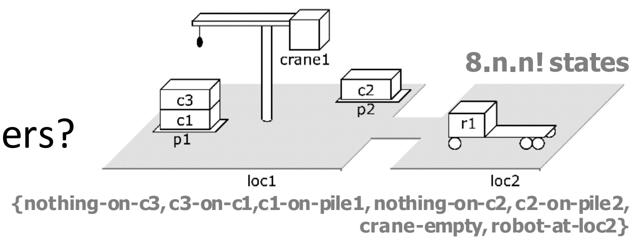
load = (
{holding,at1},
{holding},
{onrobot})

⟨take,move1,load,move2⟩ is a plan, but not a minimal plan

Set representation: properties

- **Simplicity**
  - easy to read

  How many states for n containers?



- **Computations**
  - the transition function is easy to model/compute using set operations
  - if precond(a) ⊆ s, then
    $\gamma(s,a) = (s - effects^-(a)) \cup effects^+(a)$,
- **Expressivity**
  - some sets of propositions do not describe real states
    - {holding, onrobot, at2}
  - for many domains, the set representation is still too large and not practical

Classical representation generalize the set representation by exploiting **first-order logic**.

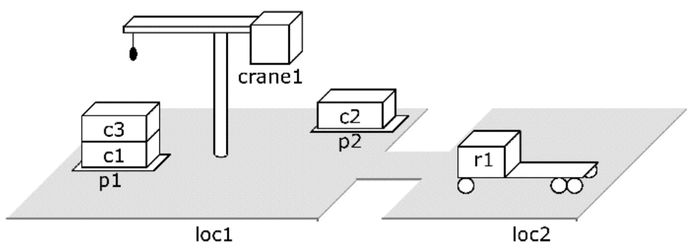
– **State** is a set of logical atoms that are true in a given state.

– **Action** is an instance of planning operator that changes true value of some atoms.

More precisely:

- L (**language**) is a finite set of predicate symbols and constants (there are no function symbols!).
- **Atom** is a predicate symbol with arguments. *example: on(c3,c1)*
- We can use **variables** in the operators. *example: on(x,y)*

State is a set of instantiated atoms (no variables). There is a finite number of states!



{attached(p1,loc1), in(c1,p1), in(c3,p1), top(c3,p1), on(c3,c1), on(c1,pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), empty(crane1), adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2), unloaded(r1)}.

– The truth value of some atoms is changing in states:
  - **fluents**
  - *example: at(r1,loc2)*

– The truth value of some state is the same in all states
  - **rigid atoms**
  - *example: adjacent(loc1,loc2)*

We will use a classical **closed world assumption**.
An atom that is not included in the state does not hold at that state!

Classical representation: planning operators

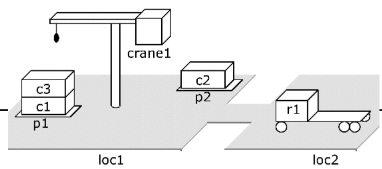
operator o is a triple (name(o), precond(o), effects(o))

- **name(o): name of the operator** in the form n(x_1,…,x_k)
  - n: a symbol of the operator (a unique name for each operator)
  - x_1,…,x_k: symbols for variables (operator parameters)
    - Must contain all variables appearing in the operator definition!
- **precond(o):**
  - literals that must hold in the state so the operator is applicable on it
- **effects(o):**
  - literals that will become true after operator application (only fluents can be there!)

take(k, l, c, d, p)
;; crane k at location l takes c off of d in pile p
precond: belong(k, l), attached(p, l), empty(k), top(c, p), on(c, d)
effects: holding(k, c), ¬ empty(k), ¬ in(c, p), ¬ top(c, p), ¬ on(c, d), top(d, p)

Classical representation: actions

An action is a fully instantiated operator
– substitute constants to variables



operator

take(k, l, c, d, p)
;; crane k at location l takes c off of d in pile p
precond: belong(k, l), attached(p, l), empty(k), top(c, p), on(c, d)
effects: holding(k, c), ¬ empty(k), ¬ in(c, p), ¬ top(c, p), ¬ on(c, d), top(d, p)

action

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

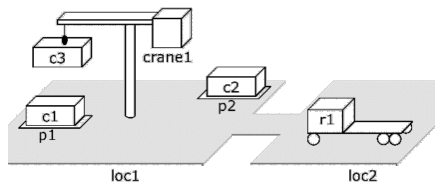
Notation:

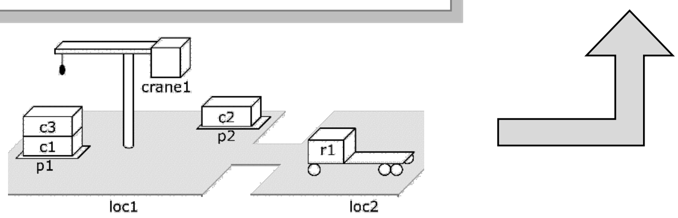
- S^+ = {positive atoms in S}
- S^- = {atoms, whose negation is in S}

Action **a** is **applicable** to state **s** if any only
$\text{precond}^+(\mathbf{a}) \subseteq \mathbf{s} \;\wedge\; \text{precond}^-(\mathbf{a}) \cap \mathbf{s} = \emptyset$

The result of application of action a to **s** is
$\gamma(\mathbf{s},\mathbf{a}) = (\mathbf{s} - \text{effects}^-(\mathbf{a})) \cup \text{effects}^+(\mathbf{a})$

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





Let L be a language and O be a set of operators.

Planning domain Σ over language L with operators O is a triple (S,A, γ):

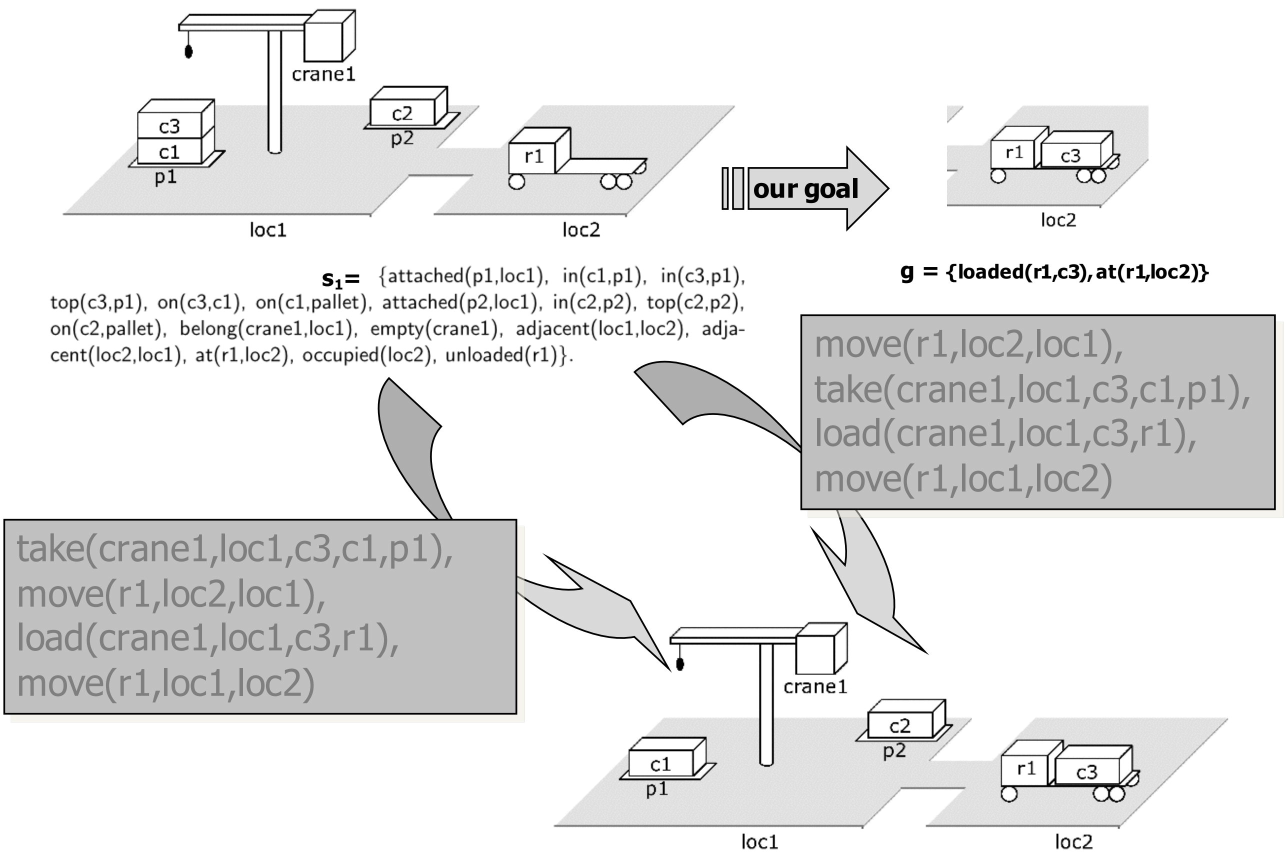
- **states** $S \subseteq 2^{\{\text{all instantiated atoms from L}\}}$
- **actions** A = {all instantiated operators from O over L}
  - action **a** is **applicable** to state **s** if $\text{precond}^+(\mathbf{a}) \subseteq \mathbf{s} \;\wedge\; \text{precond}^-(\mathbf{a}) \cap \mathbf{s} = \emptyset$
- **transition function** γ:
  - $\gamma(\mathbf{s},\mathbf{a}) = (\mathbf{s} - \text{effects}^-(\mathbf{a})) \cup \text{effects}^+(\mathbf{a})$, if **a** is applicable on **s**
  - S is closed with respect to γ (if $\mathbf{s} \in S$, then for every action **a** applicable to **s** it holds $\gamma(\mathbf{s},\mathbf{a}) \in S$)

Planning problem P is a triple ($\boldsymbol{\Sigma}$,s_0,g):

- $\boldsymbol{\Sigma}$ = (S,A, $\boldsymbol{\gamma}$) is a planning domain
- s_0 is an initial state, $s_0 \in S$
- g is a set of instantiated literals
  - state **s** satisfies the goal condition **g** if and only if $\mathbf{g}^+ \subseteq \mathbf{s} \land \mathbf{g}^- \cap \mathbf{s} = \emptyset$
  - $S_\mathbf{g} = \{\mathbf{s} \in S \mid \mathbf{s}$ satisfies $\mathbf{g}\}$ – a set of goal states

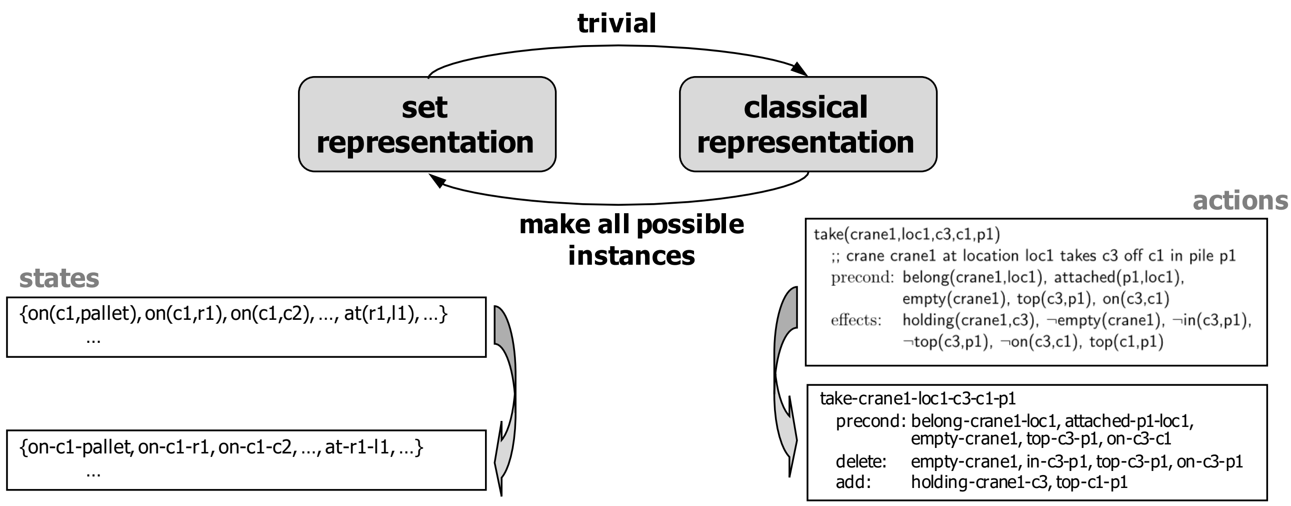
Usually the planning problem is given by a triple (O,s_0,g).

- O defines the the operators and predicates used
- s_0 provides the particular constants (objects)



Expressive power of both representations is **identical**.

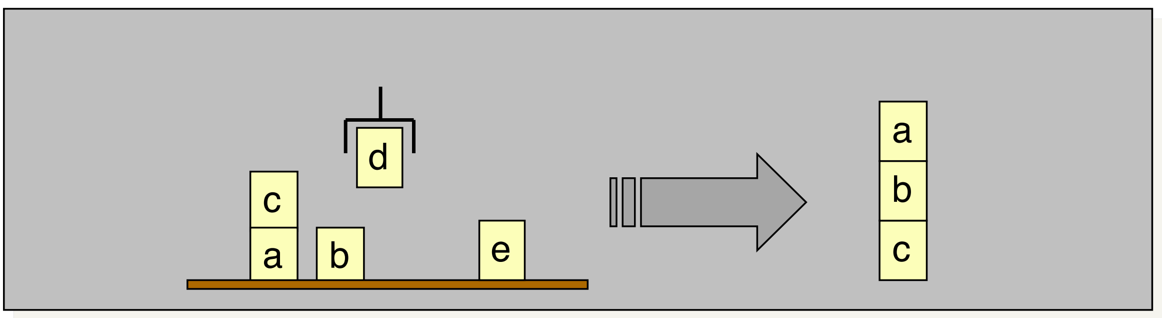
However, the translation from the classical representation to a set representation brings **exponential increase of size**.



The blocks world

- infinitely large table with a finite set of blocks
- the exact location of block on the table is irrelevant
- a block can be on the table or on another (single) block
- the planning domain deals with moving blocks by a computer hand that can hold at most one block

situation example



Blockworld: classical representation

Constants

- blocks: a,b,c,d,e

Predicates:

- **ontable(*x*)**
  block *x* is on a table
- **on(*x*,*y*)**
  block *x* is on *y*
- **clear(*x*)**
  block *x* is free to move
- **holding(*x*)**
  the hand holds block *x*
- **handempty**
  the hand is empty

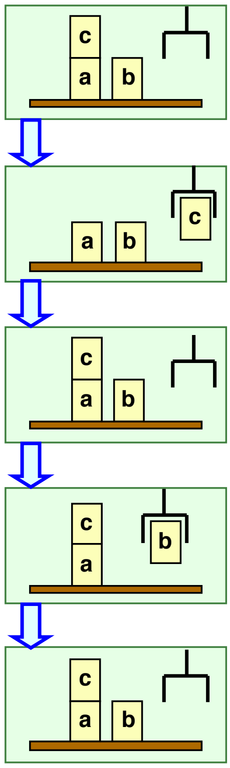
Actions

unstack(*x*,*y*)
Precond: on(*x*,*y*), clear(*x*), handempty
Effects: ¬on(*x*,*y*), ¬ clear(*x*), clear(*y*),
¬handempty, holding(*x*),

stack(*x*,*y*)
Precond: holding(*x*), clear(*y*)
Effects: ¬holding(*x*), ¬clear(*y*),
on(*x*,*y*), clear(*x*), handempty

pickup(*x*)
Precond: ontable(*x*), clear(*x*), handempty
Effects: ¬ontable(*x*), ¬clear(*x*),
¬handempty, holding(*x*)

putdown(*x*)
Precond: holding(*x*)
Effects: ¬holding(*x*), ontable(*x*),
clear(*x*), handempty



Blockworld: set representation

Propositions:
36 propositions for 5 blocks

- **ontable-a**
  block **a** is on table (5x)
- **on-c-a**
  block **c** is on block **a** (20x)
- **clear-c**
  block **c** is free to move (5x)
- **holding-d**
  the hand holds block **d** (5x)
- **handempty**
  the hand is empty (1x)

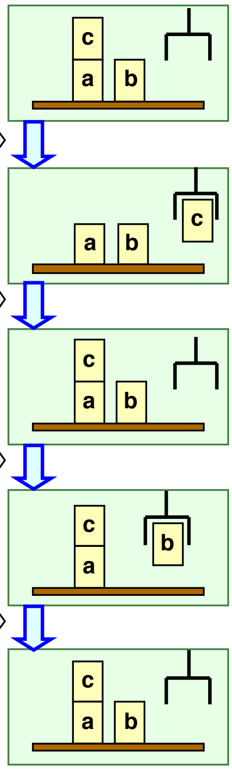
Actions
50 actions for 5 blocks

unstack-c-a
Pre: on-c-a, clear-c, handempty
Del: on-c-a, clear-c, handempty
Add: holding-c, clear-a

stack-c-a
Pre: holding-c, clear-a
Del: holding-c, clear-a
Add: on-c-a, clear-c, handempty

pickup-b
Pre: ontable-b, clear-b, handempty
Del: ontable-b, clear-b, handempty
Add: holding-b

putdown-b
Pre: holding-b
Del: holding-b
Add: ontable-b, clear-b, handempty



- **Problem Formalisation**
  - models and representations
- **State-space Planning**
  - forward and backward search
- **Plan-space Planning**
  - partial-order planning
- **Control Knowledge in Planning**
  - heuristics
  - control rules

The search space corresponds to the state space of the planning problem.

- search nodes correspond to world states
- arcs correspond to state transitions by means of actions
- the task is to find a path from the initial state to some goal state

Basic approaches

- forward search
- backward search
  - lifting
  - STRIPS
- problem dependent (blocks world)

Note: all algorithms will be presented for the classical representation

Forward planning

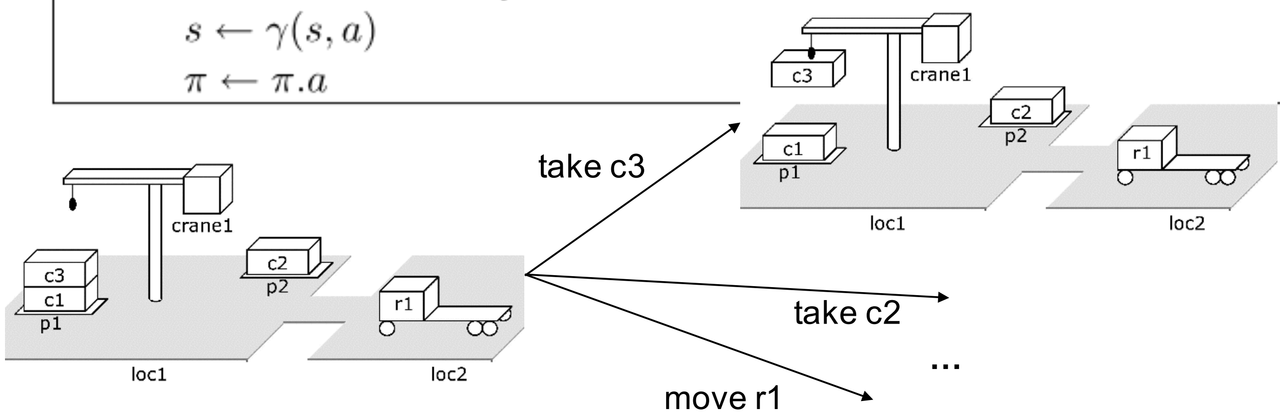
Start in the initial state and go towards some goal state.

We need to know:

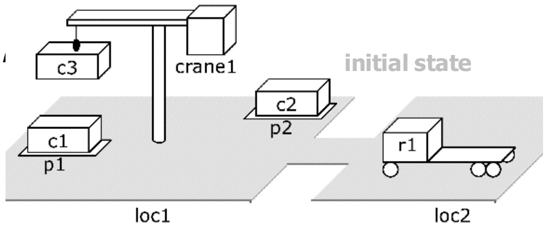
– whether a given state is a **goal state**
– how to find a set of **applicable actions** for a given state
– how to define a state after **applying a given action**

Forward planning: algorithm

Forward-search(O, s_0, g)
    $s \leftarrow s_0$
    $\pi \leftarrow$ the empty plan
    loop
        if s satisfies g then return π
        $E \leftarrow \{a | a$ is a ground instance of an operator in *O*,
                and $\mathrm{precond}(a)$ is true in $s\}$
        if $E = \emptyset$ then return failure
        nondeterministically choose an action $a \in E$
        $s \leftarrow \gamma(s, a)$
        $\pi \leftarrow \pi.a$



Forward planning: example



{belong(crane1,loc1), adjacent(loc2,loc1),
holding(crane1,c3), unloaded(r1),
at(r1,loc2), ¬occupied(loc1),
occupied(loc2),…}

move(r1,loc2,loc1)

move(r, l, m)
;; robot r moves from location l to location m
precond: adjacent(l, m), at(r, l), ¬ occupied(m)
effects: at(r, m), occupied(m), ¬ occupied(l), ¬ at(r, l)

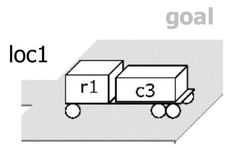
{belong(crane1,loc1),
adjacent(loc2,loc1), holding(crane1,c3), unloaded(r1),
at(r1,loc1), occupied(loc1), …}

load(crane1,loc1,c3,r1)

load(k, l, c, r)
;; crane k at location l loads container c onto robot r
precond: belong(k, l), holding(k, c), at(r, l), unloaded(r)
effects: empty(k), ¬ holding(k, c), loaded(r, c), ¬ unloaded(r)

{belong(crane1,loc1), adjacent(loc2,loc1),
empty(crane1), loaded(r1,c3),
at(r1,loc1), occupied(loc1), …}

Goal = {at(r1,loc1),loaded(r1,c3)}



Forward planning: properties

Forward planning algorithm is sound.

– If some plan is found then it is a solution plan..
– It is easy to verify by using s = $\gamma(s_0,\pi)$.

Forward planning algorithm is complete.

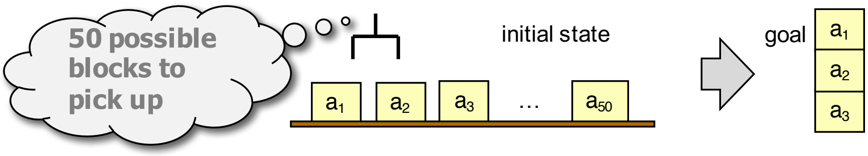
– If there is any solution plan then at least one search branch corresponds to this plan.
– induction by the plan length
– at each step, the algorithm can select the correct action from the solution plan (if correct actions were selected n the previous steps)

We need to implement the presented algorithm in a deterministic way:

- **breadth-first search**
  - sound, complete, but memory consuming
- **depth-first search**
  - sound, completeness can be guaranteed by loop checks (no state repeats at the same branch)
- **A***
  - if we have some admissible heuristic
  - the most widely used approach

What is the major problem of forward planning?

Large branching factor – the number of options



- This is a problem for deterministic algorithm that needs to explore the possible options.

Possible approaches:

- **heuristic** recommends an action to apply
- **pruning** of the search space
  - For example, if plans π_1 and π_2 reached the same state then we know that plans $\pi_1\,\pi_3$ and $\pi_2\,\pi_3$ will also reach the same state. Hence the longer of the plans π_1 and π_2 does not need to expanded.
    We need to remember the visited states ☹.

Backward planning

Start with a goal (not a goal state as there might be more goal states) and through sub-goals try to reach the initial state.

We need to know:

- whether the state **satisfies the current goal**
- how to find **relevant actions** for any goal
- how to define the **previous goal** such that the action converts it to a current goal

Backward planning: relevant actions

Action a is relevant for a goal g if and only if:

- action **a** contributes to goal **g**: $\mathbf{g} \cap \text{effects}(\mathbf{a}) \neq \emptyset$
- effects of action **a** are not conflicting goal **g**:
  - $g^- \cap \text{effects}^+(a) = \emptyset$
  - $g^+ \cap \text{effects}^-(a) = \emptyset$

A **regression set** of the goal **g** for (relevant) action **a** is

$\gamma^{-1}(g,a) = (g - \text{effects}(a)) \cup \text{precond}(a)$

Example:

goal: **{on(a,b), on(b,c)}**

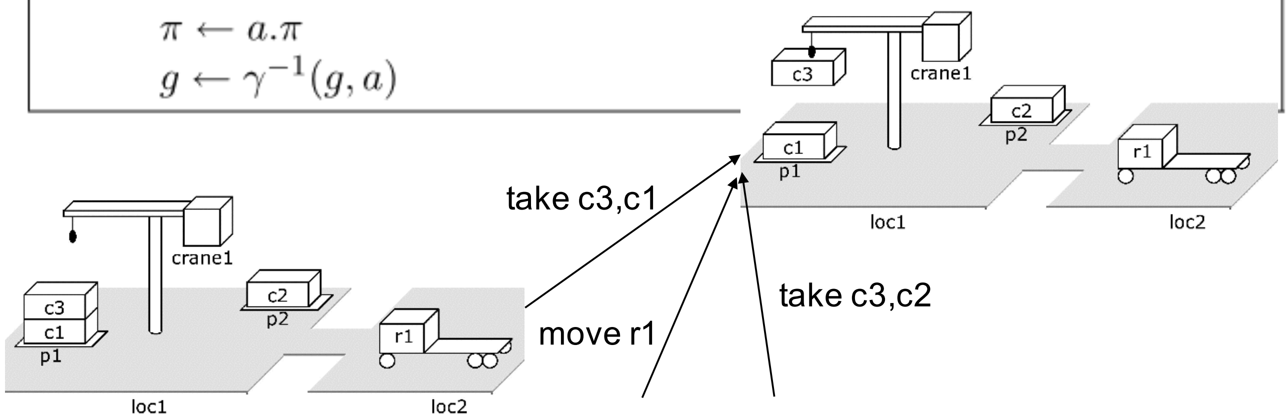
action **stack(a,b)** is relevant

> **stack(*x*,*y*)**
> Precond: holding(*x*), clear(*y*)
> Effects: ~holding(*x*), ~clear(*y*), on(*x*,*y*), clear(*x*), handempty

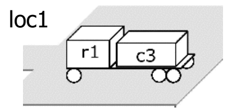
by backward application of the action we get a new goal:

{holding(a), clear(b), on(b,c)}

Backward-search(O, s_0, g)
  $\pi \leftarrow$ the empty plan
  loop
    if s_0 satisfies g then return π
    $A \leftarrow \{a|a$ is a ground instance of an operator in O
      and $\gamma^{-1}(g,a)$ is defined$\}$
    if $A = \emptyset$ then return failure
    nondeterministically choose an action $a \in A$
    $\pi \leftarrow a.\pi$
    $g \leftarrow \gamma^{-1}(g,a)$



Goal = {at(r1,loc1),loaded(r1,c3)}



load(crane1,loc1,c3,r1)

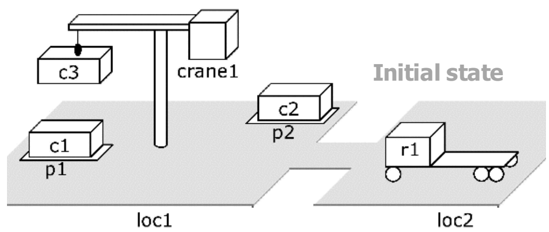
load(k, l, c, r)
  ;; crane k at location l loads container c onto robot r
  precond: belong(k, l), holding(k, c), at(r, l), unloaded(r)
  effects: empty(k), $\neg$ holding(k, c), loaded(r, c), $\neg$ unloaded(r)

{at(r1,loc1), belong(crane1,loc1),
holding(crane1,c3), unloaded(r1)}

move(r1,loc2,loc1)

move(r, l, m)
  ;; robot r moves from location l to location m
  precond: adjacent(l, m), at(r, l), $\neg$ occupied(m)
  effects: at(r, m), occupied(m), $\neg$ occupied(l), $\neg$ at(r, l)

{belong(crane1,loc1), holding(crane1,c3),
unloaded(r1),
adjacent(loc2,loc1),
at(r1,loc2),
¬occupied(loc1)}



Backward planning is **sound and complete**.

We can implement a **deterministic** version of the algorithm (via search).

- For completeness we need loop checks.
  - Let $(g_1,\ldots,g_k)$ be a sequence of goals. If $\exists i<k\ g_i \subseteq g_k$ then we can stop search exploring this branch.

Branching

- The number of options can be smaller than for the forward planning (less relevant actions for the goal).
- Still, it could be too large.
  - If we want a robot to be at the position loc51 and there are direct connections from states loc1,…,loc50, then we have 50 relevant actions. However, at this stage, the start location is not important!
  - We can instantiate actions only partially (some variables remain free. This is called **lifting**.


```
Lifted-backward-search(O, s0, g)
    π ← the empty plan
    loop
        if s0 satisfies g then return π
        A ← {(o,θ)|o is a standardization of an operator in O,
                θ is an mgu for an atom of g and an atom of effects (o),
                and γ^{-1}(θ(g),θ(o)) is defined}
        if A = ∅ then return failure
        nondeterministically choose a pair (o,θ) ∈ A
        π ← the concatenation of θ(o) and θ(π)
        g ← γ^{-1}(θ(g),θ(o))
```

Notes:

- standardization = a copy with fresh variables
- mgu = most general unifier
- by using the variables we can decrease the branching factor but the trade off is more complicated loop check

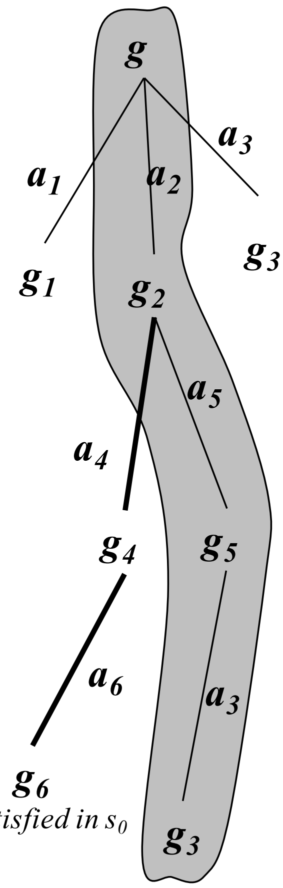
How can we further reduce the search space?

STRIPS algorithm reduces the search space of backward planning in the following way:

- **only part of the goal is assumed in each step, namely the preconditions of the last selected action**
  - instead of γ^{-1}(**s**,**a**) we can use precond(**a**) as the new goal
  - the rest of the goal must be covered later
  - This makes the algorithm incomplete!
- **If the current state satisfies the preconditions of the selected action then this action is used and never removed later upon backtracking.**

The original STRIPS algorithm is a lifted version of the algorithm below.

```
Ground-STRIPS(O, s, g)
    π ← the empty plan
    loop
        if s satisfies g then return π
        A ← {a | a is a ground instance of an operator in O,
                  and a is relevant for g}
        if A = Ø then return failure
        nondeterministically choose any action a ∈ A
        π' ← Ground-STRIPS(O, s, precond(a))
        if π' = failure then return failure
        ;; if we get here, then π' achieves precond(a) from s
        s ← γ(s, π')
        ;; s now satisfies precond(a)
        s ← γ(s, a)
        π ← π.π'.a
```



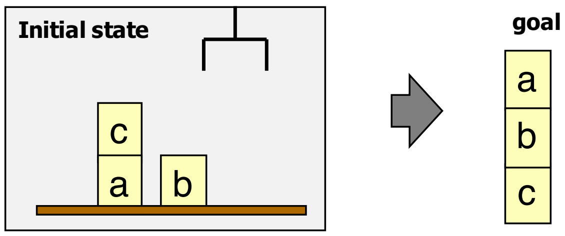
$g_2 = (g$ - effects$(a_2)) \cup$ precond(a_2)

$\pi' = \langle a_6, a_4 \rangle$ is a plan for precond(a_2)

$s = \gamma(\gamma(s_0,a_6),a_4)$ is a state satisfying precond(a_2)

Sussman anomaly is a famous example that causes troubles to the STRIPS algorithm (the algorithm can only find redundant plans).

Block world



A plan found by STRIPS may look like this:

- unstack(c,a),putdown(c),pickup(a),stack(a,b)
  now we satisfied subgoal on(a,b)
- unstack(a,b),putdown(a),pickup(b),stack(b,c)
  now we satisfied subgoal on(b,c),
  but we need to re-satisfy on(a,b) again
- pickup(a),stack(a,b) red actions can be deleted

Solving Sussman anomaly

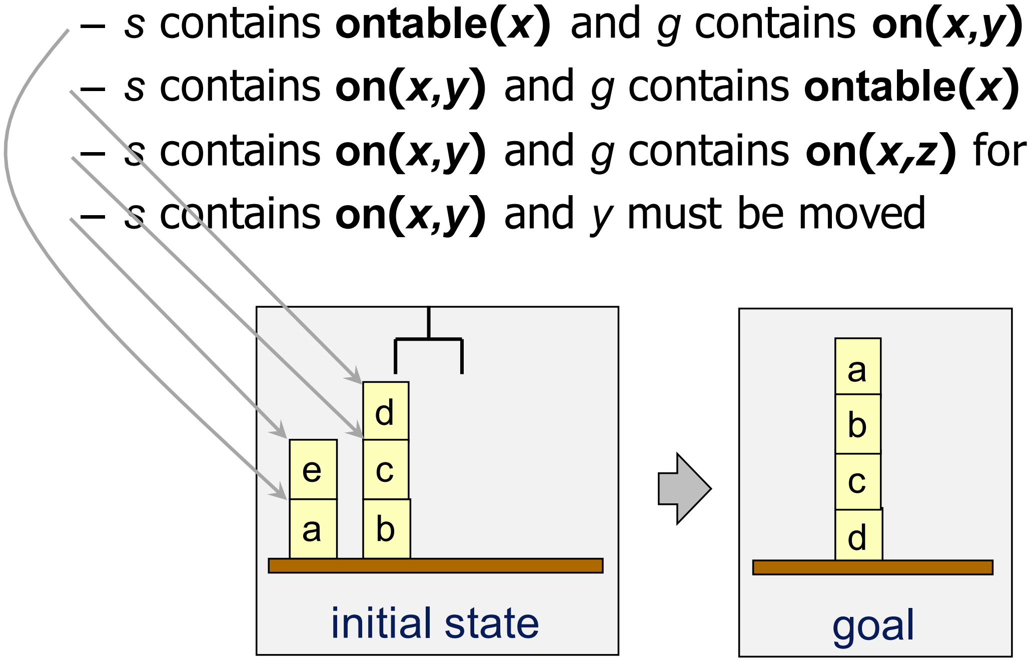
– **interleaving plans**
  - plan-space planning

– **using domain dependent information**
  - When does a solution plan exist for a blocks world?
    – all blocks from the goal are present in the initial state
    – no block in the goal stays on two other blocks (or on itself)
    – …
  - How to find a solution plan?
    Actually, it is easy and very fast!
    – put all blocks on the table (separately)
    – build the requested towers
    We can do it even better with additional knowledge!

When do we need to move block *x*?

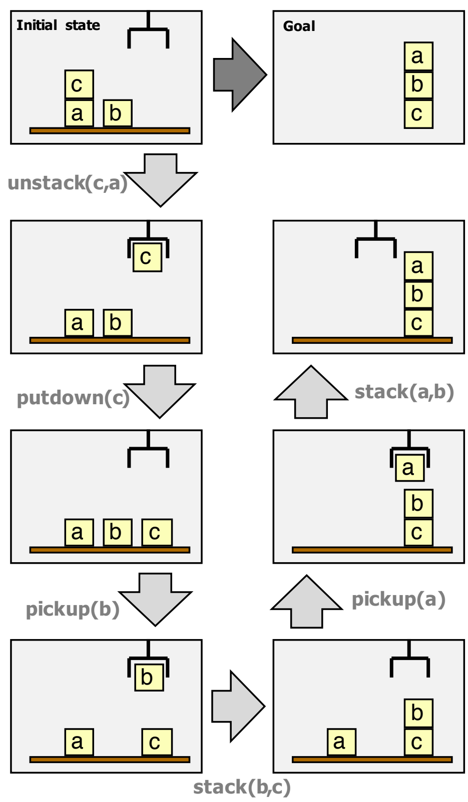
Exactly in one of the following situations:

- *s* contains **ontable(*x*)** and *g* contains **on(*x*,*y*)**
- *s* contains **on(*x*,*y*)** and *g* contains **ontable(*x*)**
- *s* contains **on(*x*,*y*)** and *g* contains **on(*x*,*z*)** for some *y*≠*z*
- *s* contains **on(*x*,*y*)** and *y* must be moved




```
Stack-containers(O, s0, g):
  if g does not satisfy the consistency conditions then
     return failure      ;; the planning problem is unsolvable
  π ← the empty plan
  s ← s0
  loop
     if s satisfies g then return π
     if there are containers b and c at the tops of their piles such that
          position(c, s) is consistent with g and on(b, c) ∈ g
     then
        append actions to π that move b to c
        s ← the result of applying these actions to s
        ;; we will never need to move b again
     else if there is a container b at the top of its pile
          such that position(b, s) is inconsistent with g
          and there is no c such that on(b, c) ∈ g
     then
        append actions to π that move b to an empty auxiliary pile
        s ← the result of applying these actions to s
        ;; we will never need to move b again
     else
        nondeterministically choose any container c such that c is
           at the top of a pile and position(c, s) is inconsistent with g
        append actions to π that move c to an empty auxiliary pallet
        s ← the result of applying these actions to s
```

Position is consistent with block *c* if there is no reason to move *c*.



- Problem Formalisation
  - models and representations
- State-space Planning
  - forward and backward search
- **Plan-space Planning**
  - partial-order planning
- Control Knowledge in Planning
  - heuristics
  - control rules

Plan space planning: core idea

The principle of plan space planning is similar to backward planning:

- start from an **„empty" plan** containing just the description of initial state and goal
- **add other actions** to satisfy not yet covered (open) goals
- if necessary **add other relations** between actions in the plan

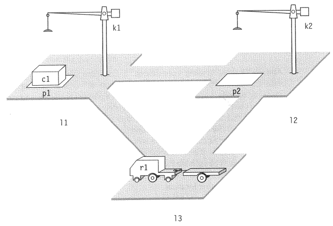
Planning is realised as **repairing flaws in a partial plan**

- go from one partial plan to another partial plan until a complete plan is found

Plan space planning: an example

Assume a partial plan with the following two actions:

- take(k1,c1,p1,l1)
- load(k1,c1,r1,l1)



Possible modifications of the plan:

- **adding a new action**
  - to apply action **load**, robot r1 must be at location l1
  - action **move**(r1,l,l1) moves robot r1 to location l1 from some location l
- **binding the variables**
  - action **move** is used for the right robot and the right location
- **ordering some actions**
  - the robot must move to the location before the action **load** can be used
  - the order with respect to action **take** is not relevant
- **adding a causal relation**
  - new action is added to move the robot to a given location that is a precondition of another action
  - the causal relation between **move** and **load** ensures that no other action between them moves the robot to another location

Plan space planning: the initial plan

The initial state and the goal are encoded using two **special actions** in the initial partial plan:

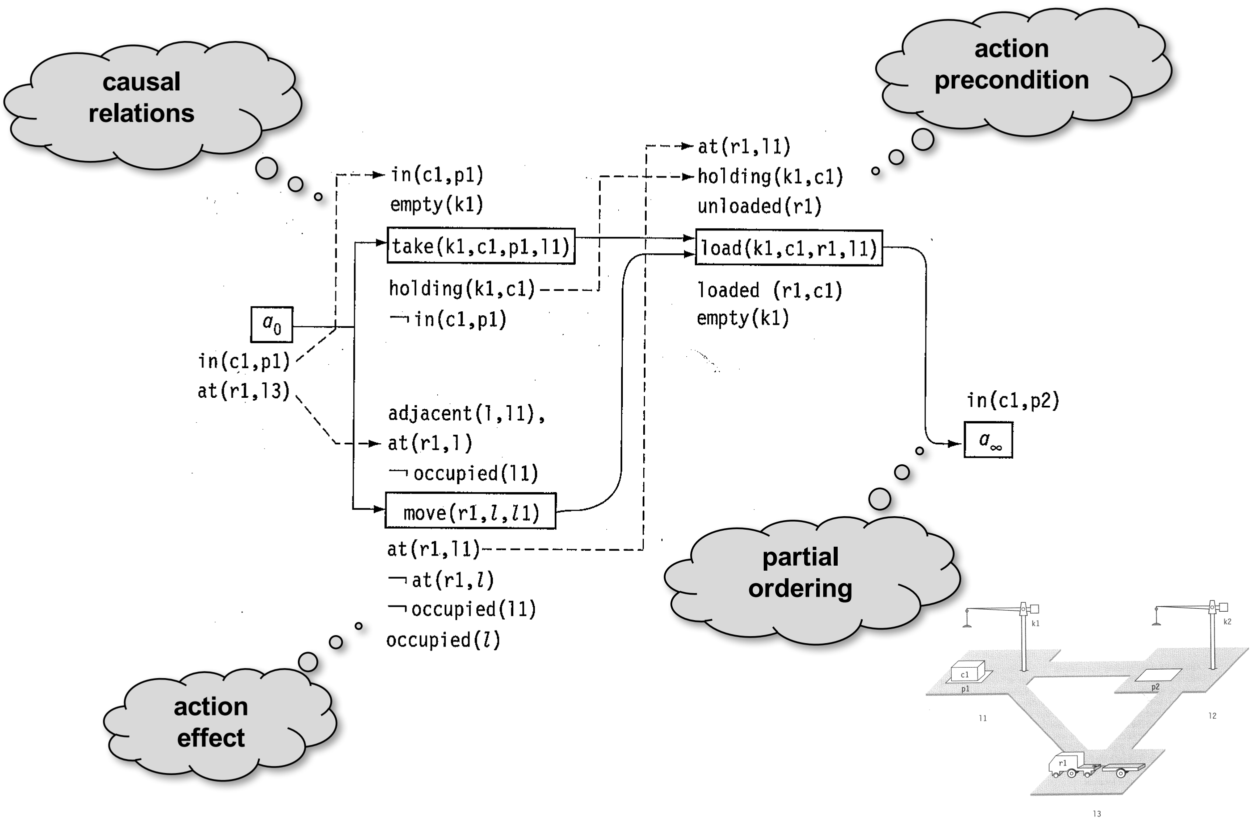
- **Action a_0 represents the initial state** in such a way that atoms from the initial state define effects of the action and there are no preconditions. This action will be before all other actions in the partial plan.
- **Action a_∞ represents the goal** in a similar way – atoms from the goal define the precondition of that action and there is no effect. This action will be after all other actions.

Planning is realised by **repairing flaws** in the partial plan.

The search nodes correspond to partial plans.

A partial plan Π is a tuple (A,<,B,L), where

- A is a set of partially instantiated planning operators $\{a_1,\ldots,a_k\}$
- $<$ is a partial order on A ($a_i<a_j$)
- B is set of constraints in the form $x=y$, $x\neq y$ or $x\in D_i$
- L is a set of causal relations ($a_i \rightarrow^p a_j$)
  - a_i,a_j are ordered actions $a_i<a_j$
  - p is a literal that is effect of a_i and precondition of a_j
  - B contains relations that bind the corresponding variables in p



Open goal is an example of a **flaw**.

This is a precondition **p** of some operator **b** in the partial plan such that no action was decided to satisfy this precondition (there is no causal relation $a_i \rightarrow^p b$).

The open goal p of action b can be resolved by:

- finding an operator **a** (either present in the partial plan or a new one) that can give **p** (**p** is among the effects of **a** and **a** can be before **b**)
- binding the variables from **p**
- adding a causal relation **a** $\rightarrow^p$**b**

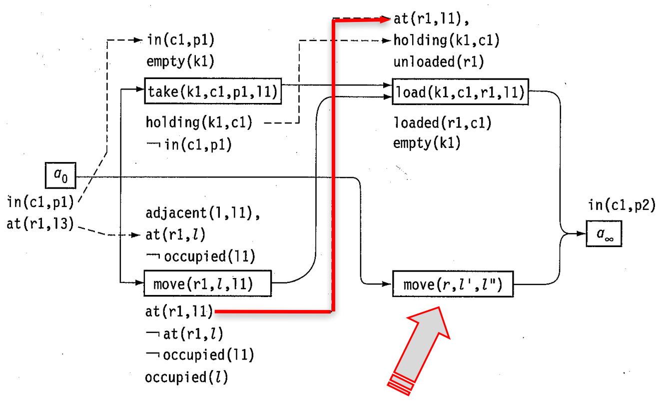
Threat is another example of **flaw**.

This is action that can influence existing causal relation.

- Let $a_i \rightarrow^p a_j$ be a causal relation and action **b** has among its effects a literal unifiable with the negation of **p** and action **b** can be between actions a_i and a_j. Then **b** is threat for that causal relation.

We can **remove the threat** by one of the ways:

- ordering **b** before $\mathbf{a_i}$
- ordering **b** after $\mathbf{a_j}$
- binding variables in **b** in such a way that **p** does not bind with the negation of **p**



Solution plan

Partial plan Π = (A,<,B,L) is a **solution plan** for the problem P = (Σ,s_0,g) if:

- partial ordering < and constraints B are globally consistent
  - there are no cycles in the partial ordering
  - we can assign variables in such a way that constraints from B hold
- Any linearly ordered sequence of fully instantiated actions from A satisfying < and B goes from s_0 to a state satisfying g.

Hmm, but this definition **does not say how** to verify that a partial plan is a solution plan!

Solution plan – a constructive view

How to efficiently verify that a partial plan is a solution plan?

Claim:

Partial plan Π = (A,<,B,L) is a solution plan if:

- there are no flaws (no open goals and no threats)
- partial ordering < and constraints B are globally consistent

Proof by induction using the plan length

- {a_0,a_1,a_∞} is a solution plan
- for more actions take one of the possible first actions and join it with action a_0

Algorithm PSP

PSP = Plan-Space Planning

```
PSP(π)
    flaws ← OpenGoals(π) ∪ Threats(π)
    if flaws = ∅ then return(π)
    select any flaw φ ∈ flaws
    resolvers ← Resolve(φ, π)
    if resolvers = ∅ then return(failure)
    nondeterministically choose a resolver ρ ∈ resolvers
    π′ ← Refine(ρ, π)
    return(PSP(π′))
end
```

Notes:

- The selection of flaw is deterministic (all flaws must be resolved).
- The resolvent is selected non-deterministically (search in case of failure).

PSP – some details

Open goals can be maintained in an **agenda** of action preconditions without causal relations. Adding a causal relation for **p** removes **p** from the agenda.

All threats can be found in time $O(n^3)$ by verifying triples of actions or threats can be maintained incrementally: after adding a new action, check causal relations influenced ($O(n^2)$), after adding a causal relation find its threats ($O(n)$).

Open goals and threats are resolved only by **consistent refinements** of the partial plan.

- consistent ordering can be detected by finding cycles or by maintaining a transitive closure of <
- consistency of constraints in B
  - If there is no negation then we can use arc consistency.
  - In case of negation, the problem of checking global consistency is NP-complete.

Properties of PSP

Algorithm PSP is **complete and sound**.

- **soundness**
  - If the algorithm finishes, it returns a consistent plan with no flaws so it is a solution plan.
- **completeness**
  - If there is a solution plan then the algorithm has the option to select the right actions to the partial plan.

Be careful about the **deterministic implementation**!

- **The search space is not finite!**
- A complete deterministic procedure must guarantee that it eventually finds a solution plan of any length – **iterative deepening** can be applied.

Algorithm PoP

PoP is a popular instance of algorithm PSP.

```
PoP(π, agenda)        ;; where π = (A, ≺, B, L)
    if agenda = ∅ then return(π)
    select any pair (a_j, p) in and remove it from agenda
    relevant ← Providers(p, π)
    if relevant = ∅ then return(failure)
    nondeterministically choose an action a_i ∈ relevant
    L ← L ∪ {⟨a_i --p--> a_j⟩}
    update B with the binding constraints of this causal link
    if a_i is a new action in A then do:
        update A with a_i
        update ≺ with (a_i ≺ a_j), (a_0 ≺ a_i ≺ a_∞)
        update agenda with all preconditions of a_i
    for each threat on ⟨a_i --p--> a_j⟩ or due to a_i do:
        resolvers ← set of resolvers for this threat
        if resolvers = ∅ then return(failure)
        nondeterministically choose a resolver in resolvers
        add that resolver to ≺ or to B
    return(PoP(π, agenda))
end
```

- **Agenda** is a set of pairs (**a**,**p**), where **p** is an open precondition of action **a**.
- **First find an action** a_i to cover some **p** from the agenda.
- **At the second stage resolve all threats** that appeared by adding action a_i or from a causal relation with a_i.

Plan-space planning: a running example

Initial state:

- At(Home), Sells(OBI,Drill), Sells(Tesco,Milk), Sells(Tesco,Banana)
- so action **Start** is defined as:
  Precond: none
  Effects: At(Home), Sells(OBI,Drill), Sells(Tesco,Milk), Sells(Tesco,Banana)

Goal:

- Have(Drill), Have(Milk), Have(Banana), At(Home)
- so action **Finish** is defined as:
  Precond: Have(Drill), Have(Milk), Have(Banana), At(Home)
  Effects: none

The following two **operators** are available:

- **Go(*l*,*m*)** ;; go from location *l* to *m*
  Precond: At(*l*)
  Effects: At(*m*), ¬At(*l*)
- **Buy(*p*,*s*)** ;; buy *p* at location *s*
  Precond: At(*s*), Sells(*s*,*p*)
  Effects: Have(*p*)

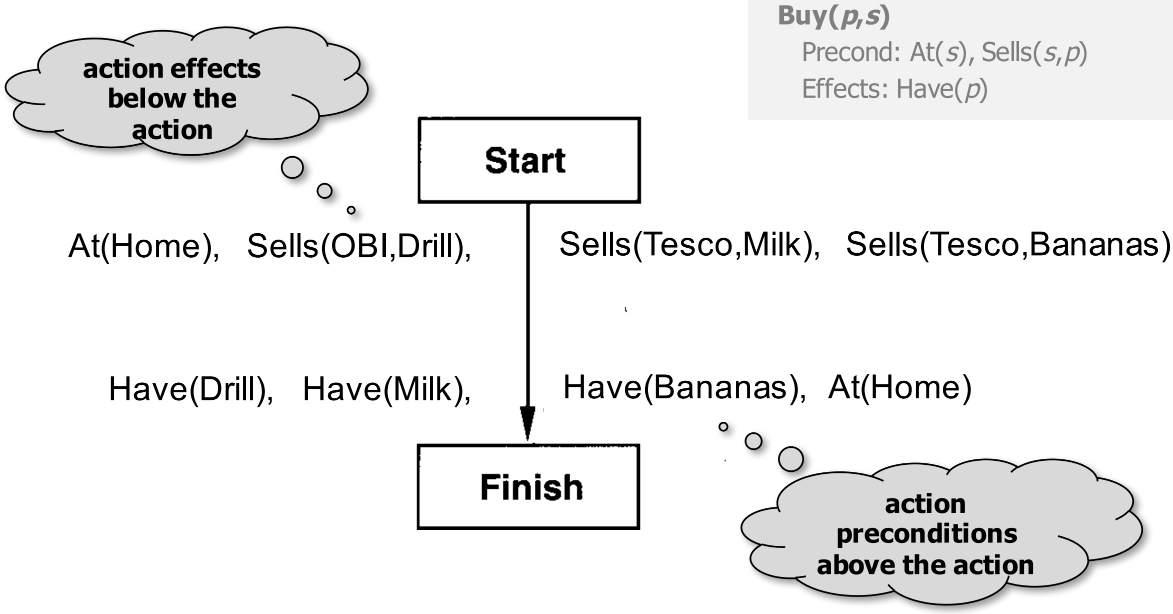
Plan-space planning: a running example

The initial (empty) plan

Operators

Go(*l*,*m*)
Precond: At(*l*)
Effects: At(*m*), ¬At(*l*)

Buy(*p*,*s*)
Precond: At(*s*), Sells(*s*,*p*)
Effects: Have(*p*)



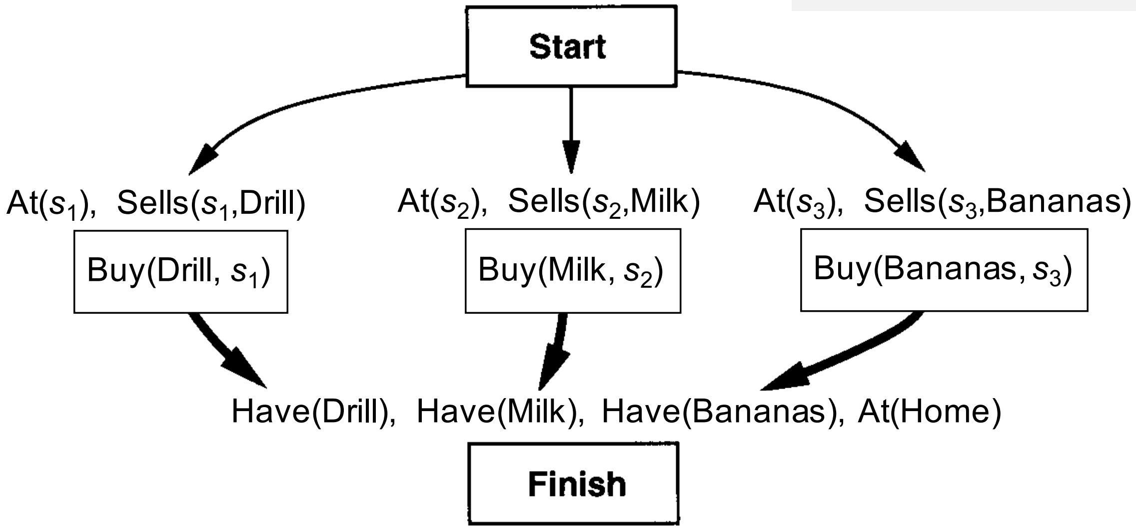
Plan-space planning: a running example

There is only one way to satisfy the **open goals Have**, and this is via **actions Buy** (no threats added).

Operators

Go(*l*,*m*)
Precond: At(*l*)
Effects: At(*m*), ¬At(*l*)

Buy(*p*,*s*)
Precond: At(*s*), Sells(*s*,*p*)
Effects: Have(*p*)



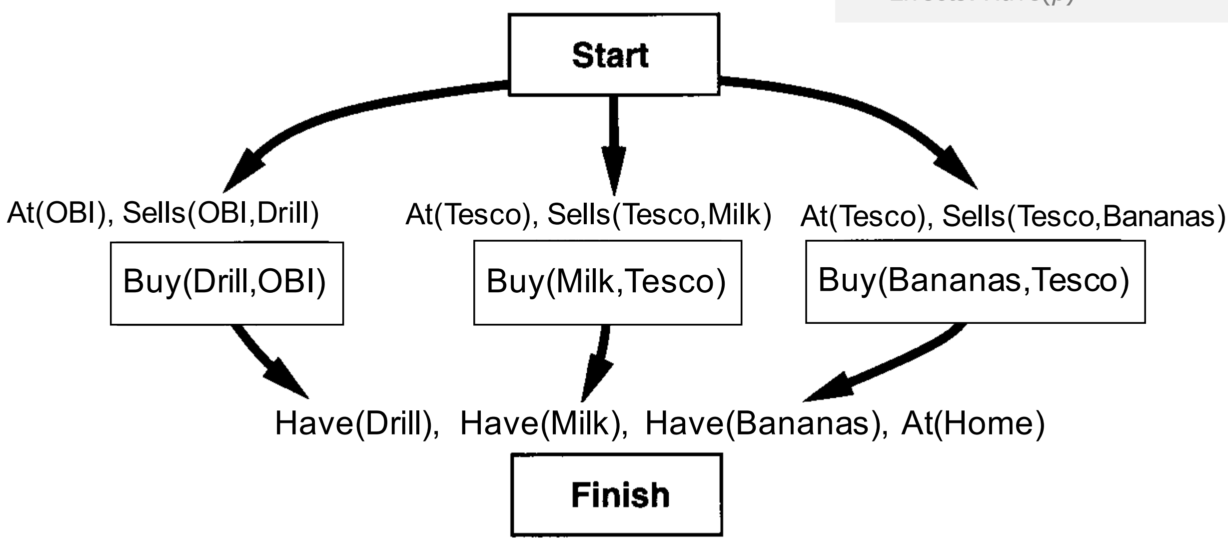
Plan-space planning: a running example

There is again a single way to satisfy preconditions **Sells** and this is substituting the right **constants.**

Operators

Go(*l*,*m*)
Precond: At(*l*)
Effects: At(*m*), ¬At(*l*)

Buy(*p*,*s*)
Precond: At(*s*), Sells(*s*,*p*)
Effects: Have(*p*)



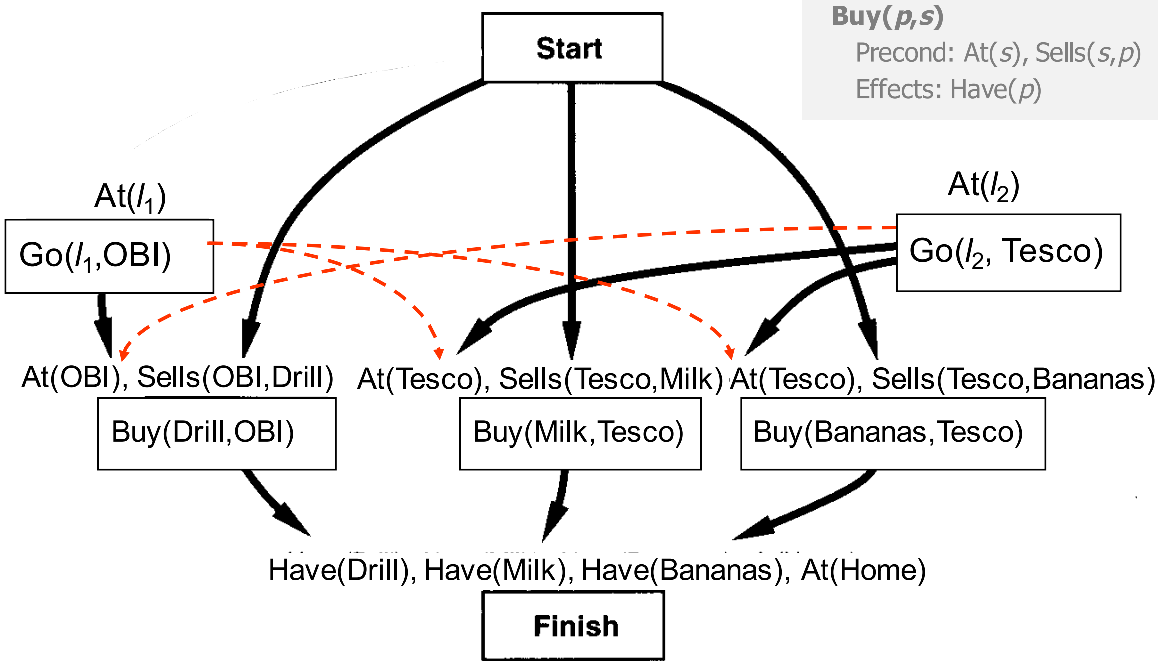
The only way to **satisfy open goals** is by adding actions **Go.**

- There are new threats!

Operators

Go(*l*,*m*)
Precond: At(*l*)
Effects: At(*m*), ¬At(*l*)

Buy(*p*,*s*)
Precond: At(*s*), Sells(*s*,*p*)
Effects: Have(*p*)



Plan-space planning: a running example

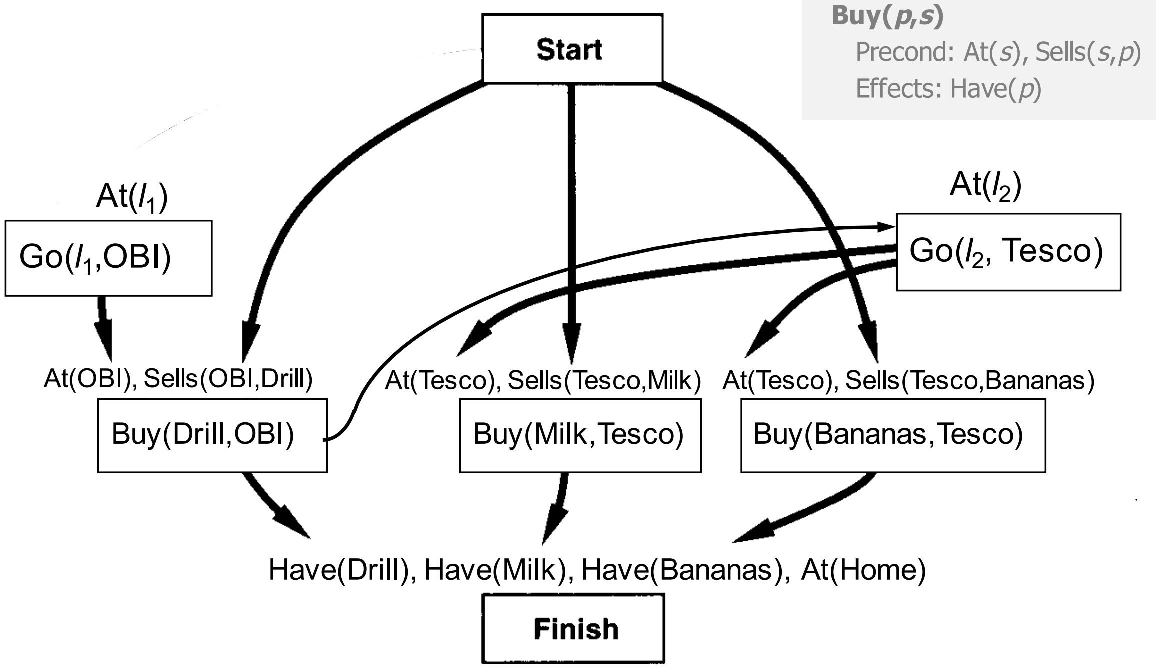
One **threat** can be solved by ordering Buy(Drill) before Go(Tesco)

- This solves all the threats!

Operators

Go(*l*,*m*)
Precond: At(*l*)
Effects: At(*m*), ¬At(*l*)

Buy(*p*,*s*)
Precond: At(*s*), Sells(*s*,*p*)
Effects: Have(*p*)



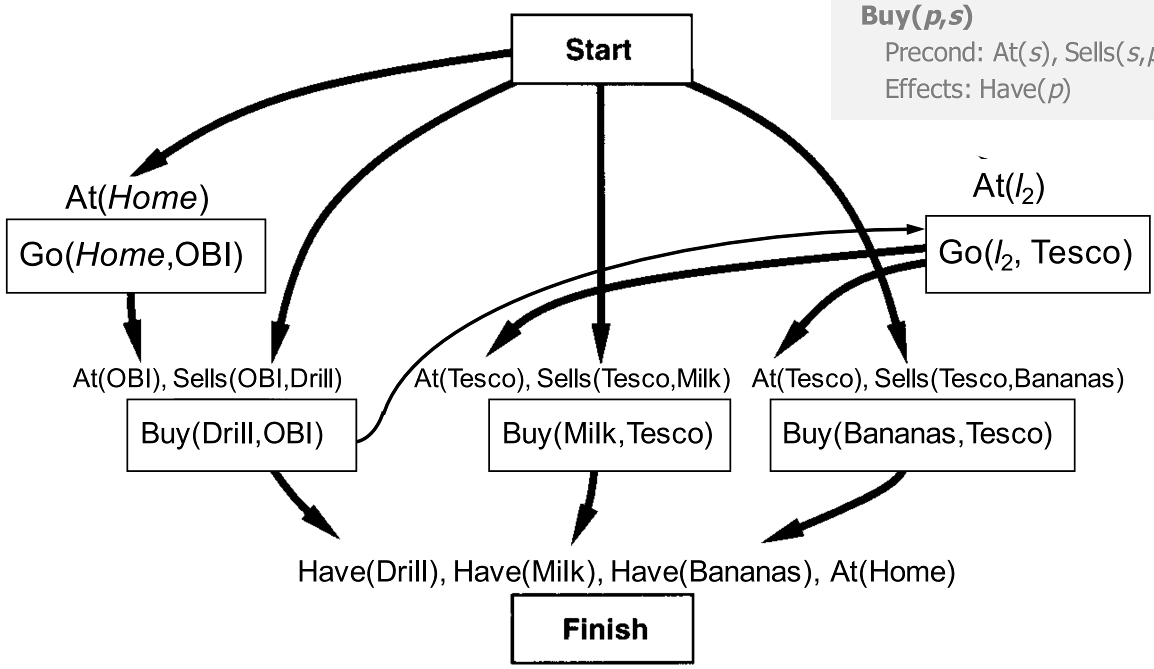
Plan-space planning: a running example

Open goal At(l_1) can be satisfied by **assignment** l_1=Home taken from the action Start.

Operators

Go(*l*,*m*)
Precond: At(*l*)
Effects: At(*m*), ¬At(*l*)

Buy(*p*,*s*)
Precond: At(*s*), Sells(*s*,*p*)
Effects: Have(*p*)



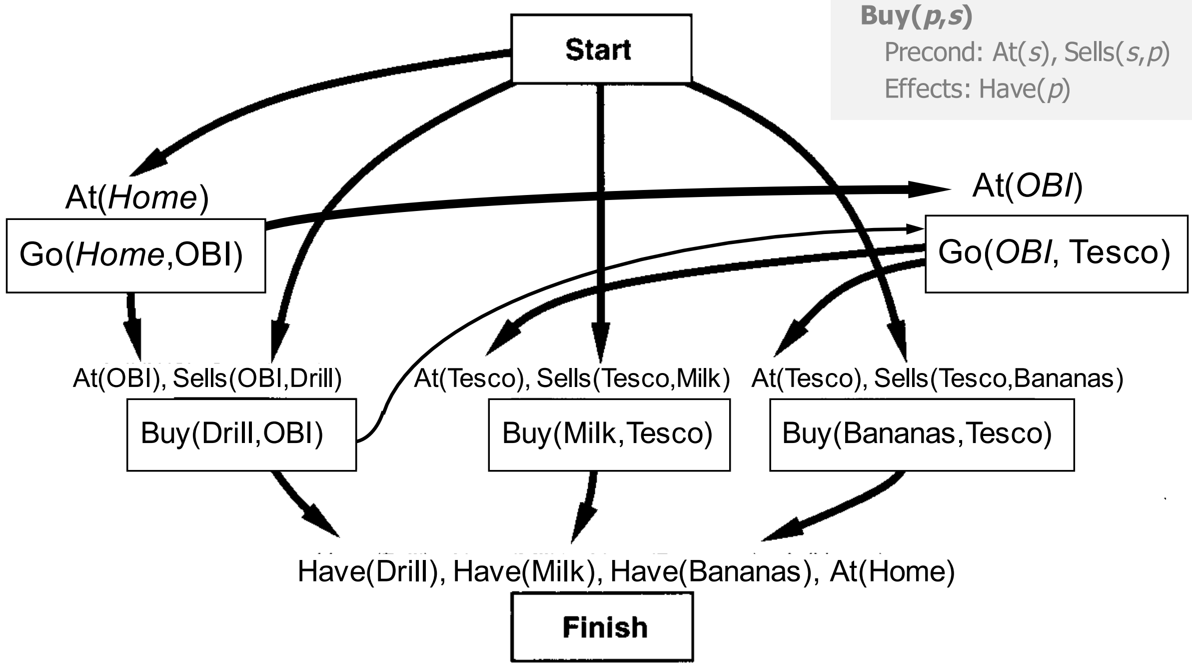
Plan-space planning: a running example

Open goal At(l_2) can be satisfied by **assignment** l_2=OBI from action Go(Home, OBI)

Operators

Go(*l*,*m*)
Precond: At(*l*)
Effects: At(*m*), ¬At(*l*)

Buy(*p*,*s*)
Precond: At(*s*), Sells(*s*,*p*)
Effects: Have(*p*)



Plan-space planning: a running example

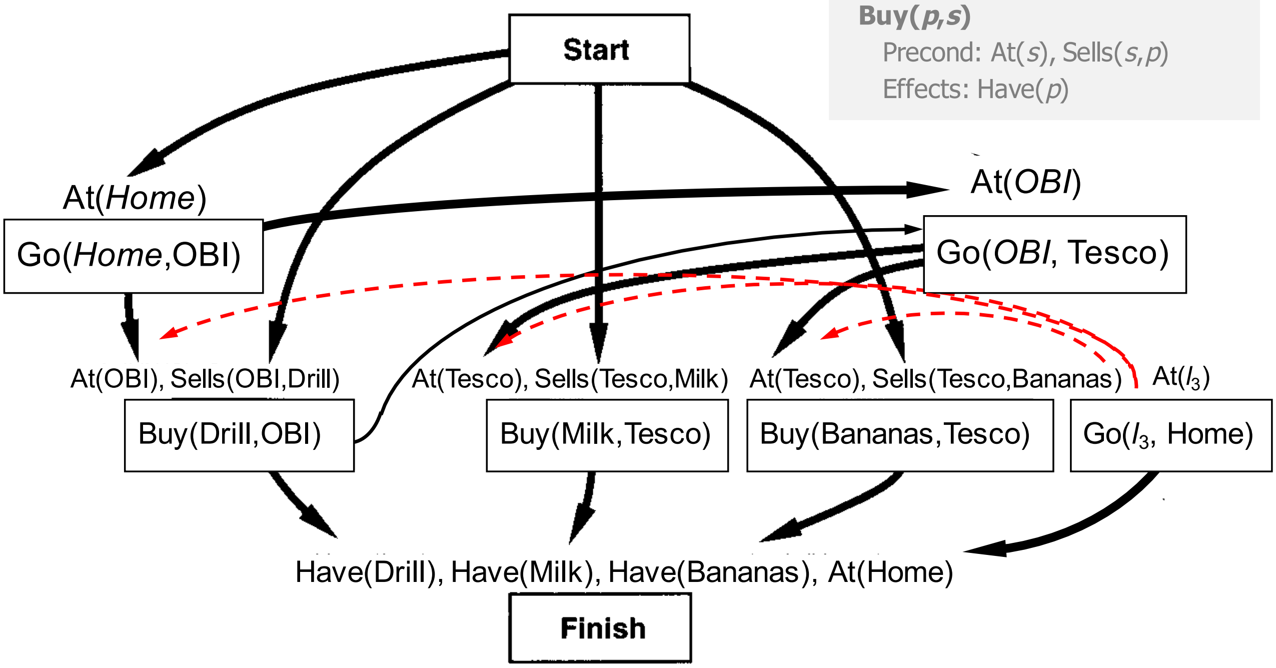
Open goal At(Home) from Finish is satisfied by **action** Go

– new threats appear

Operators

Go(*l*,*m*)
Precond: At(*l*)
Effects: At(*m*), ¬At(*l*)

Buy(*p*,*s*)
Precond: At(*s*), Sells(*s*,*p*)
Effects: Have(*p*)



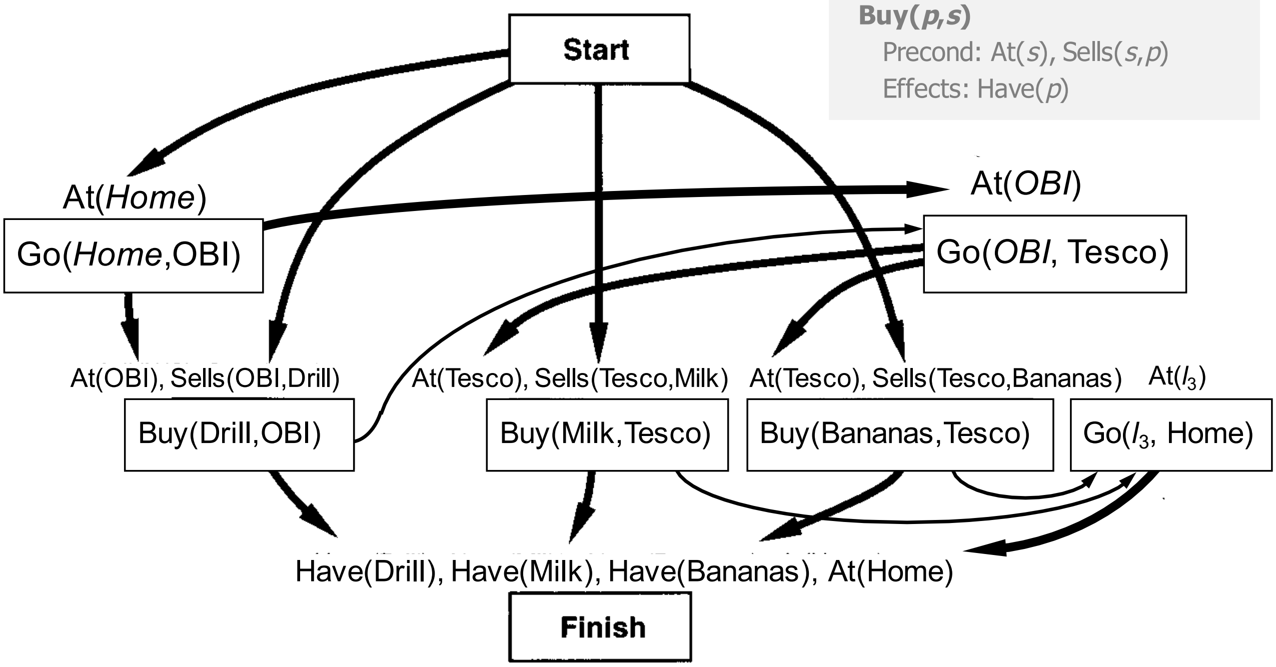
Plan-space planning: a running example

Threats for At(Tesco) are removed by **ordering** Go(Home) after both actions Buy

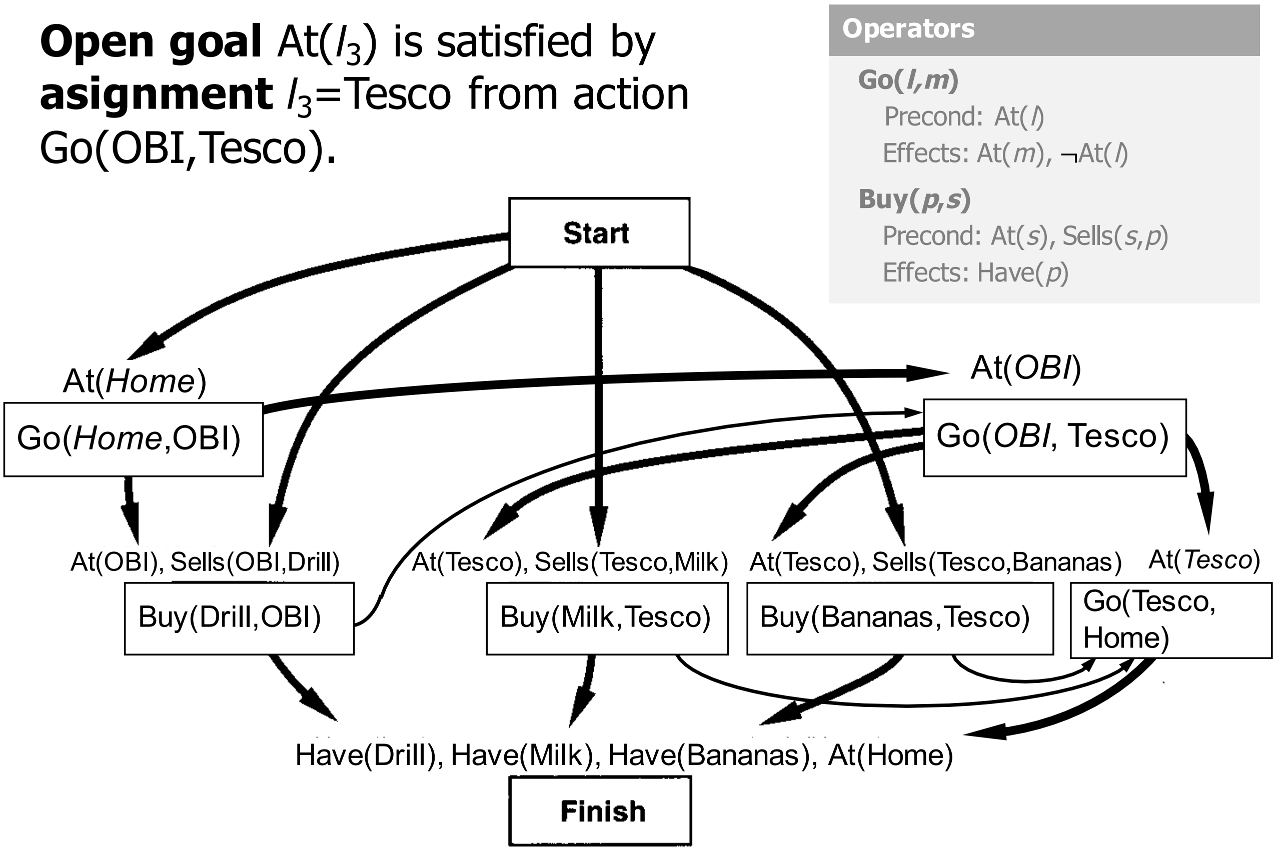
Operators

Go(*l*,*m*)
Precond: At(*l*)
Effects: At(*m*), ¬At(*l*)

Buy(*p*,*s*)
Precond: At(*s*), Sells(*s*,*p*)
Effects: Have(*p*)



Open goal At(l_3) is satisfied by **asignment** l_3=Tesco from action Go(OBI,Tesco).



Comparison

	State space planning	Plan space planning
search space	finite	infinite
search nodes	simple (world states)	complex (partial plans)
world states	explicit	not used
partial plan	action selection and ordering done together	action selection and ordering separated
plan structure	linear	causal relations

State space planning is much faster today thanks to heuristics based on state evaluation.

However, **plan space planning**:

- makes more **flexible plans** thanks to partial order
- supports **further extensions** such as adding explicit time and resources

- **Problem Formalisation**
  - models and representations
- **State-space Planning**
  - forward and backward search
- **Plan-space Planning**
  - partial-order planning
- **Control Knowledge in Planning**
  - heuristics
  - control rules

Heuristics are used to select next search node to be explored (recall, that we described the planning algorithms using non-determinism).

- Note: If we know, which node to select to get a solution, then we use **oracle**. With oracle we will find the solution deterministically.

Naturally, we prefer the heuristic to be as **close** as possible **to oracle** while being **computed efficiently**.

A typical way to obtain (admissible) heuristics is via solving a **relaxed problem** (some problem constraints are relaxed – not assumed).

- solve the relaxed problem for the successor nodes
- select the node with the best solution of the relaxed problem

For optimisation problems the heuristic h(u) estimates the real cost h*(u) of the best solution reachable via node u.

- the heuristic is **admissible**, if $h(u) \leq h^*(u)$ (for minimization)
- the search algorithms using admissible heuristics are optimal

State-space heuristics

Heuristic estimates the **number of actions** to reach a goal state from a given state or to reach a given predicate or a set of predicates.

Based on solving a **“relaxed” problem**:

- assume only positive effects
- assume that different atoms can be reached independently

Zero attempt:

- $\Delta_0(s,p) = 0$ if $p \in s$
- $\Delta_0(s,g) = 0$ if $g \subseteq s$
- $\Delta_0(s,p) = \infty$ if $p \notin s$ and $\forall a \in A$, $p \notin effects^+(a)$
- $\Delta_0(s,p) = \min_a\{1+ \Delta_0(s,precond(a)) \mid p \in effects^+(a)\}$
- $\Delta_0(s,g) = \Sigma_{p \in g}\, \Delta_0(s,p)$

This heuristic is **not admissible** (for optimal planning) because it does not provide a lower bound for the plan length!

Delta(s)
    for each p do: if $p \in s$ then $\Delta_0(s,p) \leftarrow 0$, else $\Delta_0(s,p) \leftarrow \infty$
    $U \leftarrow s$
    iterate
        for each a such that precond(a) $\subseteq U$ do
            $U \leftarrow U \cup$ effects$^+(a)$
            for each $p \in$ effects$^+(a)$ do
                $\Delta_0(s,p) \leftarrow \min\{\Delta_0(s,p), 1 + \sum_{q \in \text{precond}(a)} \Delta_0(s,q)\}$
    until no change occurs in the above updates
end

State-space admissible heuristics

A first attempt to admissible heuristic

- …
- $\Delta_1(s,g)= \max\{\Delta_0(s,p) \mid p \in g\}$
- If the heuristic value is greater than the best so-far solution then we can cut-off the search branch.
- Based on experiments, heuristic Δ_1 is less informed than Δ_0.

A second attempt to admissible heuristic

Let us try to explore reachability of pairs of atoms together.

- …
- $\Delta_2(s,p)=\min_a\{1+ \Delta_2(s,precond(a)) \mid p \in effects^+(a)\}$
- $\Delta_2(s,\{p,q\})=\min\{$
  $\min_a\{1+ \Delta_2(s,precond(a)) \mid \{p,q\} \subseteq effects^+(a)\}$,
  $\min_a\{1+ \Delta_2(s,\{q\} \cup precond(a)) \mid p \in effects^+(a)\}$,
  $\min_a\{1+ \Delta_2(s,\{p\} \cup precond(a)) \mid q \in effects^+(a)\}\}$
- $\Delta_2(s,g)= \max_{p,q}\{\Delta_2(s,\{p,q\}) \mid \{p,q\} \subseteq g\}$

We can generalise the above idea to larger sets of atoms, but for k>2 this heuristic is computationally expensive.

State-space planning with heuristics

Forward planning

- Prefer the action leading to a state with smaller heuristic distance to a goal.
- Heuristic is computed in every search step.

```
Heuristic-forward-search(π, s, g, A)
    if s satisfies g then return π
    options ← {a ∈ A | a applicable to s}
    for each a ∈ options do Delta(γ(s, a))
    while options ≠ ∅ do
        a ← argmin{Δ0(γ(s, a), g) | a ∈ options}
        options ← options − {a}
        π' ← Heuristic-forward-search(π. a, γ(s, a), g, A)
        if π' ≠ failure then return(π')
    return(failure)
end
```

Backward planning

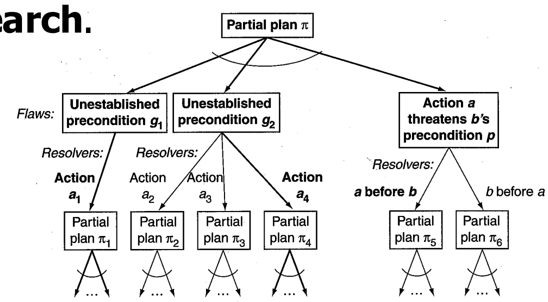
- First, compute the heuristic distance from the initial state s_0 to all atoms: $\Delta(s_0,p)$
  – can be done incrementally
- Prefer the action whose regression set is heuristically closer to the initial state.

```
Backward-search(π, s0, g, A)
    if s0 satisfies g then return(π)
    options ← {a ∈ A | a relevant for g}
    while options ≠ ∅ do
        a ← argmin{Δ0(s0, γ^-1(g, a)) | a ∈ options}
        options ← options − {a}
        π' ← Backward-search(a. π, s0, γ^-1(g, a), A)
        if π' ≠ failure then return(π')
    return failure
end
```

Plan-space heuristics

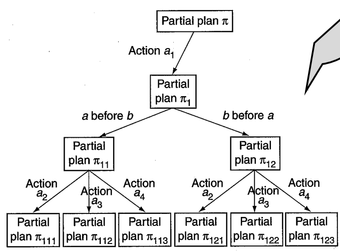
Plan-space planning is based on **AND-OR search**.
There are two types of choices:

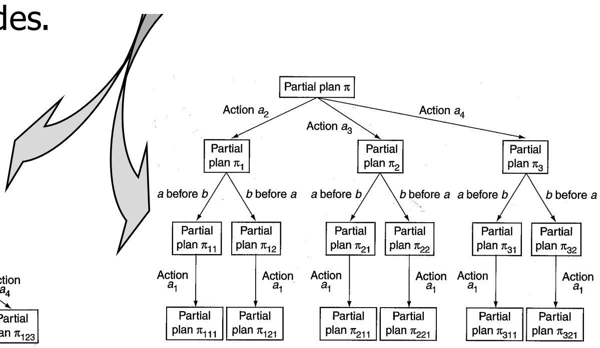
– the choice of flaw (AND node)
– the choice of resolver (OR node)



Flaw-selection heuristic

– This is a form of **serialization of the AND/OR** tree, in particular the AND node is split into several nodes.
– Which serialization is better?





– Better serialization leads to a smaller number of nodes in the graph.
– **FAF (fewest alternatives first) heuristic**
  - first repair the flaws with fewer ways for repair

Resolver-selection heuristic

Which resolver for a flaw should be tried first?

Let $\{\pi_1, \ldots, \pi_m\}$ be partial plans obtained by applying different flaw resolvers and g_π be a set of open goals in π.

- **Zero attempt**
  prefer a partial plan with fewer open goals
  ⇨ $\eta_0(\pi) = |g_\pi|$
  – However, this does not really estimate the size of the plan.
- **Next attempt**
  Generate an AND-OR graph for π till given depth k and count the number of new actions and the number of open goals not in s_0
  ⇨ $\eta_k(\pi)$
  – This is **too computationally expensive**.
- **One more improvement**
  Construct a planning graph (once) for the original goal. Then find an open goal p in π, that was added last to the graph and on the path from s_0 to p count the number of actions that are not in π
  ⇨ $\eta(\pi)$

Pruning

Heuristics guide the planner towards a goal state by ordering alternative plans. They do not solve the problem with the **large number of alternatives**.

Can we **detect and prune bad alternatives**?

Example (blockworld)

- If a block is placed correctly (consistent with the goal) then any action that moves that block just enlarges the plan.
- If a block is on a wrong place and there is an action that moves it to the correct place then any action that moves the block elsewhere just enlarges the plan.

Domain dependent information can prune the search space, but the open question is how to express such information for a general planning algorithm.

- **control rules**

We need a formalism to express relations between the current world state and future states.

Simple temporal logic

- extension of first-order logic by **modal operators**
  - $\phi_1 \cup \phi_2$ (until) — ϕ_1 is true in all states until the first state (if any) in which ϕ_2 is true
  - $\square\ \phi$ (always) — ϕ is true now and in all future states
  - $\diamond\ \phi$ (eventually) — ϕ is true now or in any future state
  - $\bigcirc\ \phi$ (next) — ϕ is true in the next state
  - GOAL(ϕ) — ϕ (no modal operators) is true in the goal state
- ϕ is a logical formula expressing relations between the objects of the world (it can include modal operators)

The **interpretation** of modal formula involves not just the current state but we need to work with a triple **(S, s_i, g)**:

- $S = \langle s_0, s_1, \dots \rangle$ is an infinite sequence of states
- $s_i \in S$ is the current state
- g is a goal formula

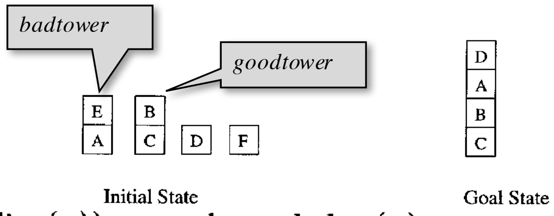
Plan $\pi = \langle a_1, a_2, \dots, a_n \rangle$ gives a finite sequence of states $S_\pi = \langle s_0, s_1, \dots, s_n \rangle$ where $s_{i+1} = \gamma(s_i, a_{i+1})$, that can be made infinite $\langle s_0, s_1, \dots, s_{n-1}, s_n, s_n, s_n, \dots \rangle$

$(S, s_i, g) \models \phi$ is defined as follows:

- $(S, s_i, g) \models \phi$ iff $s_i \models \phi$ for atom ϕ
- $(S, s_i, g) \models \phi_1 \wedge \phi_2$ iff $(S, s_i, g) \models \phi_1$ a $(S, s_i, g) \models \phi_2$
- …
- $(S, s_i, g) \models \phi_1 \cup \phi_2$ iff there exists $j \geq i$ st. $(S, s_j, g) \models \phi_2$ and for each k: $i \leq k < j$ $(S, s_k, g) \models \phi_1$
- $(S, s_i, g) \models \square\ \phi$ iff $(S, s_j, g) \models \phi$ for each $j \geq i$
- $(S, s_i, g) \models \diamond\ \phi$ iff $(S, s_j, g) \models \phi$ for some $j \geq i$
- $(S, s_i, g) \models \bigcirc\ \phi$ iff $(S, s_{i+1}, g) \models \phi$
- $(S, s_i, g) \models$ GOAL(ϕ) iff $\phi \in g$

Control rules: an example

Goodtower is a tower such that no block needs to be moved.
Badtower is a tower that is not good.



$$goodtower(x) \triangleq clear(x) \land \neg\text{GOAL}(holding(x)) \land goodtowerbelow(x)$$

$$\begin{aligned} goodtowerbelow(x) \triangleq\ & (ontable(x) \land \neg\exists[y{:}\text{GOAL}(on(x,y))]\,)) \\ & \lor \exists[y{:}on(x,y)]\ \neg\text{GOAL}(ontable(x)) \land \neg\text{GOAL}(holding(y)) \land \neg\text{GOAL}(clear(y)) \\ & \land \forall[z{:}\text{GOAL}(on(x,z))]\ z = y \land \forall[z{:}\text{GOAL}(on(z,y))]\ z = x \\ & \land goodtowerbelow(y) \end{aligned}$$

$$badtower(x) \triangleq clear(x) \land \neg goodtower(x)$$

Control rule:

$$\begin{aligned} \Box\big(& \forall[x{:}clear(x)]\ goodtower(x) \Rightarrow \bigcirc(clear(x) \lor \exists[y{:}on(y,x)]\ goodtower(y)) \\ & \land badtower(x) \Rightarrow \bigcirc(\neg\exists[y{:}on(y,x)]\,) \\ & \land (ontable(x) \land \exists[y{:}\text{GOAL}(on(x,y))]\ \neg goodtower(y)) \\ & \quad \Rightarrow \bigcirc(\neg holding(x))\big) \end{aligned}$$

goodtower remains *goodtower*

do not put anything on *badtower*

do not take a block from a table until you can put it on a *goodtower*

Progression

To use control rules in planning we need to express how the formula changes when we **go from state s_i to state s_{i+1}**.

– We look for a formula progr(ϕ, s_i) that is true in s_{i+1}, if ϕ is true in state s_i

- ϕ does not contain any modal operator
  – progr(ϕ, s_i) = true if $s_i \vdash \phi$
    = false if $s_i \vdash \phi$ does not hold
- ϕ with logical connectives
  – progr($\phi_1 \land \phi_2$, s_i) = progr(ϕ_1, s_i) $\land$ progr(ϕ_2, s_i)
  – progr($\neg\, \phi$, s_i) = $\neg$progr(ϕ, s_i)
- ϕ with quantifiers (no function symbols, just k constants c_j)
  – progr($\forall x\ \phi$, s_i) = progr($\phi\{x/c_1\}$, s_i) $\land \ldots \land$ progr($\phi\{x/c_k\}$, s_i)
  – progr($\exists\ x\ \phi$, s_i) = progr($\phi\{x/c_1\}$, s_i) $\lor \ldots \lor$ progr($\phi\{x/c_k\}$, s_i)
- ϕ with modal operators
  – progr($\phi_1 \cup \phi_2$, s_i) = (($\phi_1 \cup \phi_2$) $\land$ progr (ϕ_1, s_i)) $\lor$ progr (ϕ_2, s_i)
  – progr($\Box\, \phi$, s_i) = ($\Box\ \phi$) $\land$ progr(ϕ, s_i)
  – progr($\Diamond\, \phi$, s_i) = ($\Diamond\ \phi$) $\lor$ progr(ϕ, s_i)
  – progr($\bigcirc\, \phi$, s_i) = ϕ

Technical notes:

– progress(ϕ, s_i) is obtained from progr(ϕ, s_i) by cleaning (true $\land$ d $\rightarrow$ d, $\neg$true $\rightarrow$ false, …)
– Can be extended to a sequence of states $\langle s_0, \ldots, s_n \rangle$
progress(ϕ, $\langle s_0, \ldots, s_n \rangle$) = ϕ if n = 0
= progress(progress(ϕ, $\langle s_0, \ldots, s_{n-1} \rangle$), s_n) otherwise

Properties of progression

(S,s_i,g) ⊨ ϕ iff (S,s_{i+1},g) ⊨ progress(ϕ, s_i).

- i.e. progress behaves as we need

(S,s_0,g) ⊨ ϕ then for any prefix S‘ = ⟨s_0, … ,s_i⟩ of S it holds progress(ϕ,S‘) ≠ false.

- If the control rule is satisfied then progress is not false

If plan π is applicable to s_0 and progress(ϕ, S_π) = false, then there is no extension S‘ of S_π st. (S‘,s_0,g) ⊨ ϕ

- If progress is false then the control rule cannot be satisfied

> The planning algorithm will modify the control rule for next states by applying progress and if progress is false then we know that there is no plan (going through a given state) satisfying the control rule.

Planning with control rules

Forward state-space planning guided by control rules.

- If a partial plan S_π violates the control rule progress(ϕ, S_π), then the plan is not expanded.

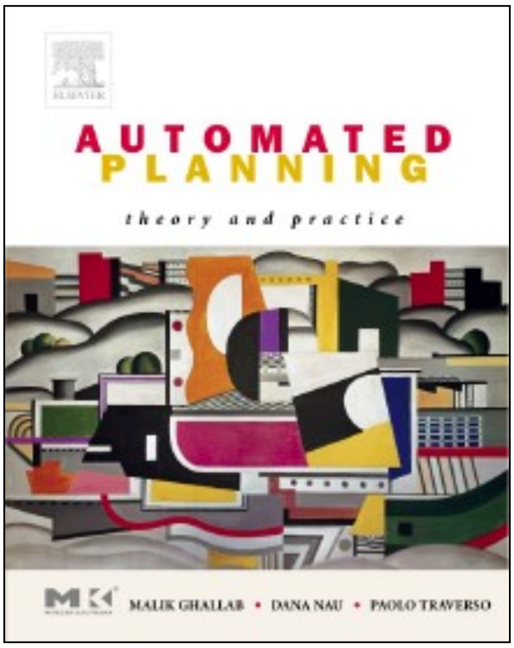
```
STL-plan(O, s0, g, φ)
    s ← s0
    π ← the empty plan
    loop
        if φ = false then return failure
        if s satisfies g then return π
        A ← {a | a is a ground instance of an operator in O
                  and precond(a) is true in s}
        if A = ∅ then return failure
        nondeterministically choose an action a ∈ A
        s ← γ(s, a)
        π ← π.a
        φ ← progress(φ, s)
```

- if ϕ = false then return failure: a partial plan violates the control rule ϕ
- if s satisfies g then return π: a complete plan found
- A ← {…}: actions applicable to state s
- s ← γ(s, a): a new state after the action
- ϕ ← progress(ϕ, s): control rule progression ϕ

Conclusions

- What we did not cover:
  - State-variable representation
  - Problem solving by transformation to SAT/CSP
  - Hierarchical task networks
  - Planning with time and resources
  - Planning with uncertainty and dynamic worlds
- What we have learned:
  - Formalization of planning problems
  - Mainstream solving approaches

Textbook



Automated Planning: Theory and Practice

- M. Ghallab, D. Nau, P. Traverso
- http://www.laas.fr/planning/
- Morgan Kaufmann

Charles University in Prague, Faculty of Mathematics and Physics
bartak@ktiml.mff.cuni.cz