Planning & Scheduling

Roman Barták
Department of Theoretical Computer Science and Mathematical Logic

State Space Planning

Just to recall

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

- Σ is a **planning domain** describing possible world states and actions (state transitions)
- s_0 is an **initial state**
- g describes the **goal states**

Set representation

- **state** is as set of propositions
- **action** is a triple of proposition sets (precond,effects$^-$,effects$^+$)
  $\text{precond} \subseteq s \rightarrow (s - \text{effects}^-) \cup \text{effects}^+$

Classical representation

- **state** is a set of instantiated atoms
- **operator** is a triple (name, precond, effects), where precond and effects are sets of literals
- **action** is a fully instantiated operator
  $\text{precond}^+ \subseteq s \ \wedge \ \text{precond}^- \cap s = \emptyset \rightarrow (s - \text{effects}^-) \cup \text{effects}^+$

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

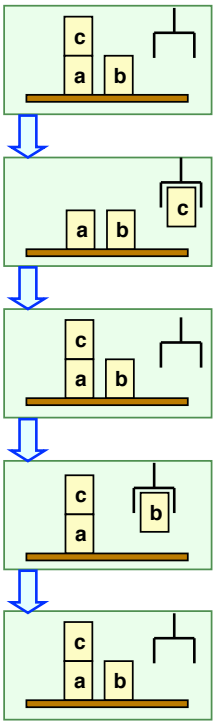
Actions

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)

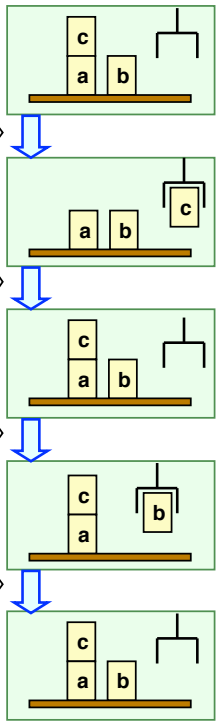
Actions
50 actions for 5 blocks

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



- What is the complexity to solve a planning problem in the classical representation?
- **Decidability**

Function symbols	Plan existence	Plan of given length
no	yes	yes
yes	partially	yes

Halting problem

- **Complexity**

Negative effects	Negative preconditions	Plan existence	Plan of given length
yes	yes/no	EXPSPACE-c	NEXPTIME-c
no	yes	NEXPTIME-c	NEXPTIME-c
	no	EXPTIME-c	NEXPTIME-c

- Almost all planning algorithms are based on some form of **search**.
- The algorithms differ in the search space to be explored and in the way of exploration.
  - **State Space Planning**
    - search nodes directly correspond to world states
  - **Plan Space Planning**
    - search nodes correspond to partial plans

- **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

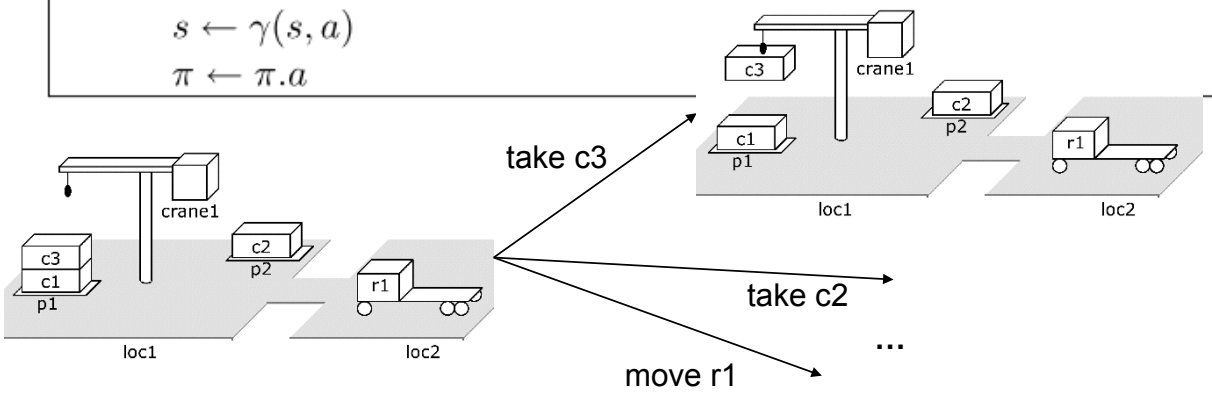
Start in the initial state and go towards some goal state.

We need to know:

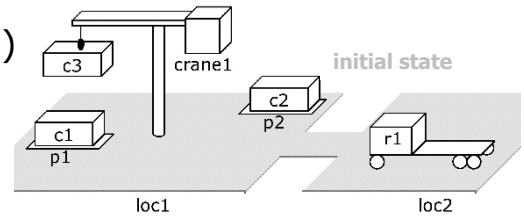
- whether a given state is a **goal state**
- how to find a set of **applicable actions** for a given state
- how to define a state after **applying a given actio**n

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 $\text{precond}(a)$ 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), $\neg$occupied(m)
effects: at(r, m), occupied(m), $\neg$occupied(l), $\neg$at(r, l)

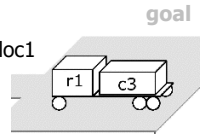
{belong(crane1,loc1), adjacent(loc2,loc1), holding(crane1,c3), unloaded(r1), at(r1,loc1), occupied(loc1), ...}

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), $\neg$holding(k, c), loaded(r, c), $\neg$unloaded(r)

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

Cíl = {at(r1,loc1),loaded(r1,c3)}



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 in the previous steps)

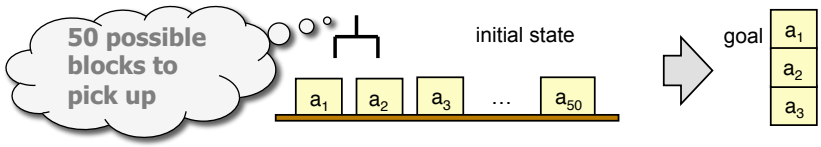
We need to implement the presented algorithm in a deterministic way:

- **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

Branching

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**: **g** ∩ effects(**a**) ≠ ∅
- effects of action **a** are not conflicting goal **g**:
  - g^- ∩ $effects^+(a)$ = ∅
  - g^+ ∩ $effects^-(a)$ = ∅

A **regression set** of the goal **g** for (relevant) action **a** is
$\gamma^{-1}(g,a) = (g - effects(a)) \cup precond(a)$

Example:

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

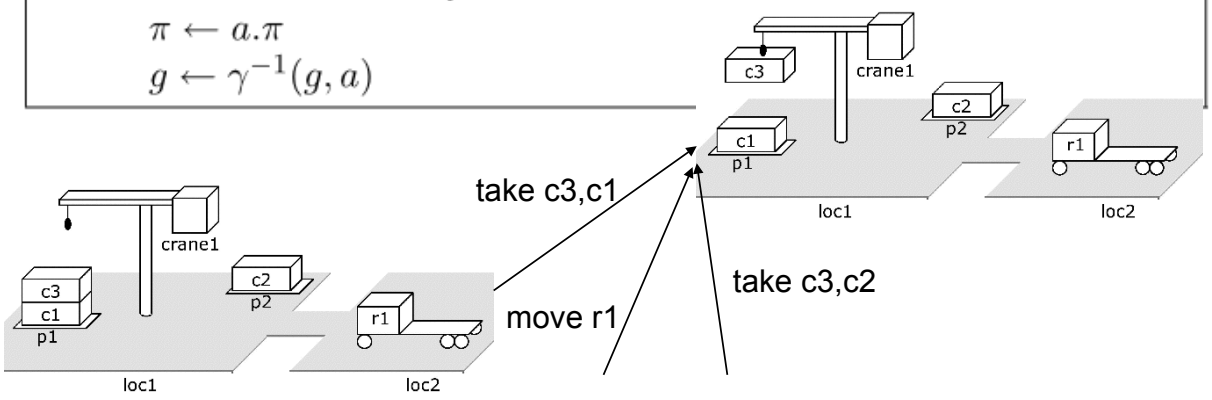
action **stack(a,b)** is relevant

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)}

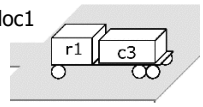
Backward planning: algorithm

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 $a \in A$
    $\pi \leftarrow a.\pi$
    $g \leftarrow \gamma^{-1}(g,a)$



Backward planning: an example

Cíl = {at(r1,loc1),loaded(r1,c3)}



load(crane1,loc1,c3,r1)

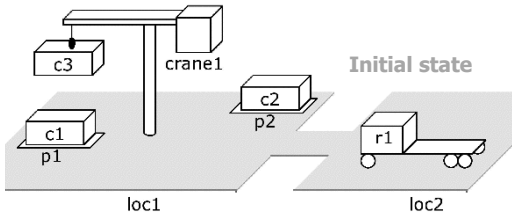
load(k, l, c, r)
;; crane k at location l loads container c onto robot r
precond: belong(k, l), holding(k, c), at(r, l), unloaded(r)
effects: empty(k), ¬ holding(k, c), loaded(r, c), ¬ unloaded(r)

{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), ¬ occupied(m)
effects: at(r, m), occupied(m), ¬ occupied(l), ¬ at(r, l)

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



Backward planning: properties

- 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**.

Backward planning: a lifted version

Lifted-backward-search(O, s_0, g)
  $\pi \leftarrow$ the empty plan
  loop
    if s_0 satisfies g then return π
    $A \leftarrow \{(o, \theta) | o$ is a standardization of an operator in O,
      θ is an mgu for an atom of g and an atom of effects (o),
      and $\gamma^{-1}(\theta(g), \theta(o))$ is defined$\}$
    if $A = \emptyset$ then return failure
    nondeterministically choose a pair $(o, \theta) \in A$
    $\pi \leftarrow$ the concatenation of $\theta(o)$ and $\theta(\pi)$
    $g \leftarrow \gamma^{-1}(\theta(g), \theta(o))$

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

STRIPS

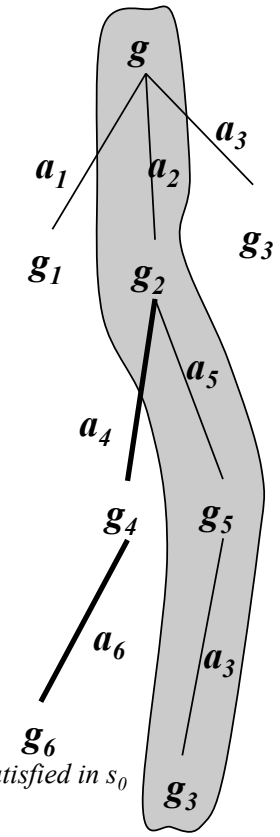
- How can we further reduce the search space?
- **STRIPS algorithm** reduces the search space of backward planning in the following way:
  - **only part of the goal is assumed in each step, namely the preconditions of the last selected action**
    - instead of γ^{-1}(**s**,**a**) we can use precond(**a**) 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.**

STRIPS algorithm

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

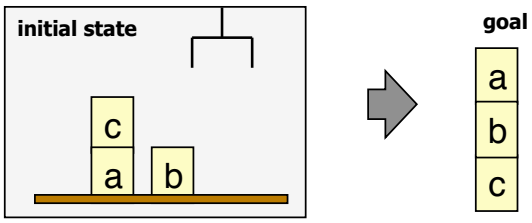
```
Ground-STRIPS(O, s, g)
    π ← the empty plan
    loop
        if s satisfies g then return π
        A ← {a | a is a ground instance of an operator in O,
                  and a is relevant for g}
        if A = ∅ then return failure
        nondeterministically choose any action a ∈ A
        π′ ← 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
```

$g_2 = (g - \text{effects}(a_2)) \cup \text{precond}(a_2)$
$\pi' = \langle a_6, a_4 \rangle$ is a plan for $\text{precond}(a_2)$
$s = \gamma(\gamma(s_0, a_6), a_4)$ is a state satisfying $\text{precond}(a_2)$



Sussman anomaly

- 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?

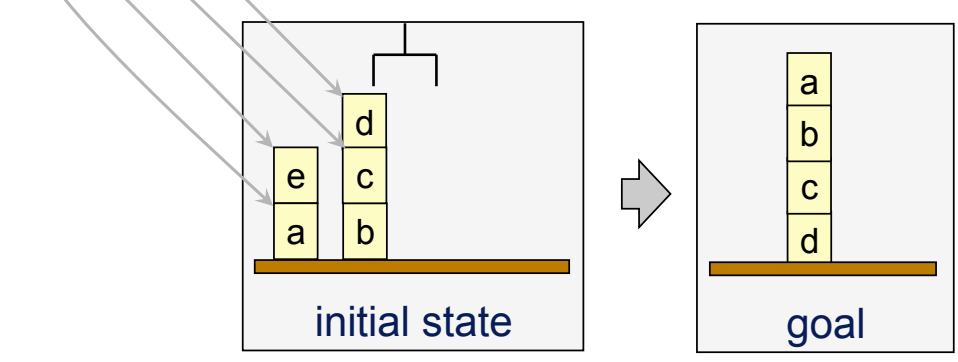
      Actually, it is easy and very fast!
      - put all blocks on the table (separately)
      - build the requested towers

      We can do it even better with additional knowledge!

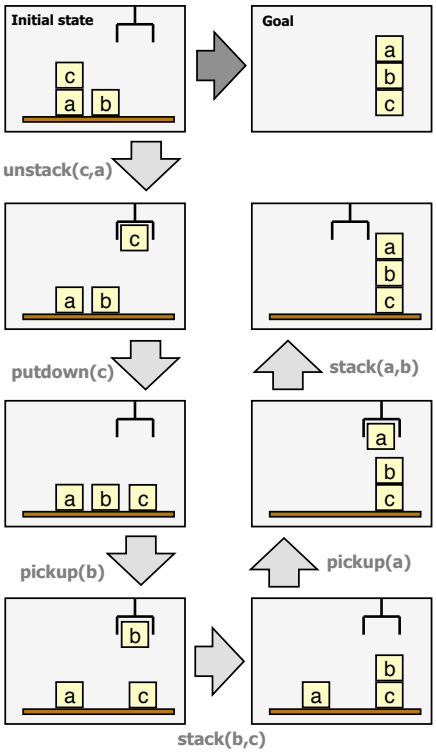
When do we need to move block *x*?

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 $y \neq z$
- *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*.

Department of Theoretical Computer Science and Mathematical Logic
bartak@ktiml.mff.cuni.cz