Foundations of Automated Planning

Roman Barták

Charles University, Czech Republic

So far, we studied general planning algorithms. Now we will look at several approaches for improving the efficiency of planning.

- **heuristics**
  - domain independent search guide
- **control rules**
  - domain dependent pruning
- **hierarchical task networks**
  - domain dependent recipes

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

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 ∈ s then Δ0(s,p) ← 0, else Δ0(s,p) ← ∞
    U ← s
    iterate
        for each a such that precond(a) ⊆ U do
            U ← U ∪ effects+(a)
            for each p ∈ effects+(a) do
                Δ0(s,p) ← min{Δ0(s,p), 1 + Σ_{q∈precond(a)} Δ0(s,q)}
    until no change occurs in the above updates
end
```

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.

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

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

What about the Graphplan?

- The above principles resemble the expansion stage of Graphplan, but Graphplan also provides mutexes.
- Heuristic Δ_2 can be modified to be closer to Graphplan by assuming reachability of two atoms by independent actions as a single step

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
```

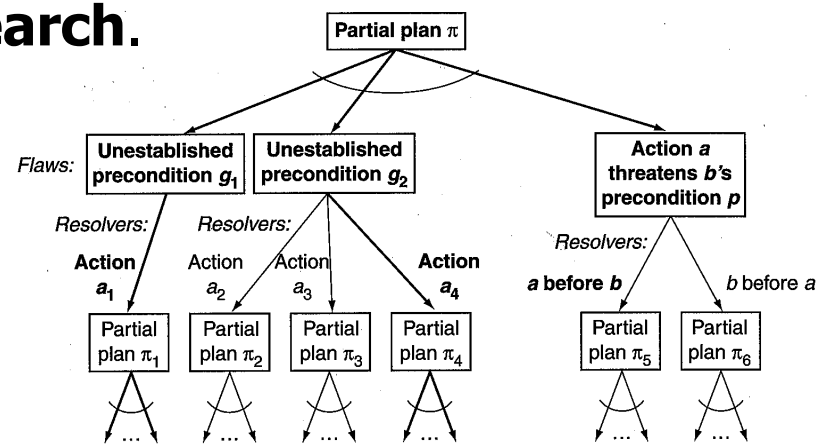
Backward planning

- 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
```

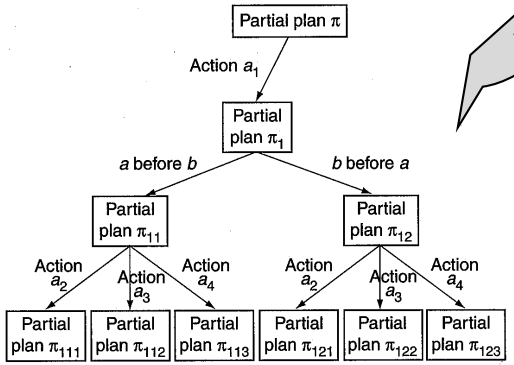
Plan-space planning is based on **AND-OR search**.
There are two types of choices:

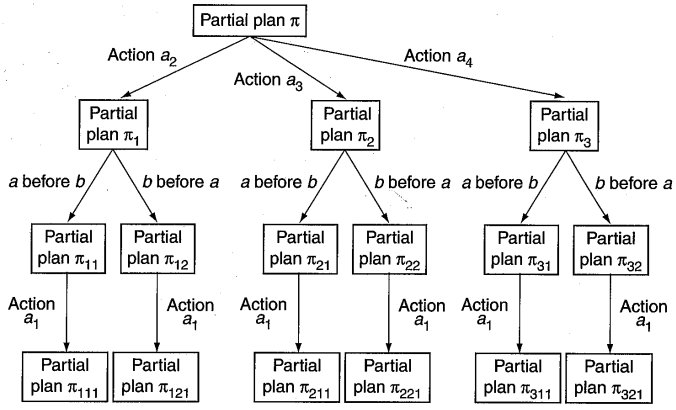
- the choice of flaw (AND node)
- the choice of resolver (OR node)



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

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
  ⇨ $\eta(\pi)$

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.

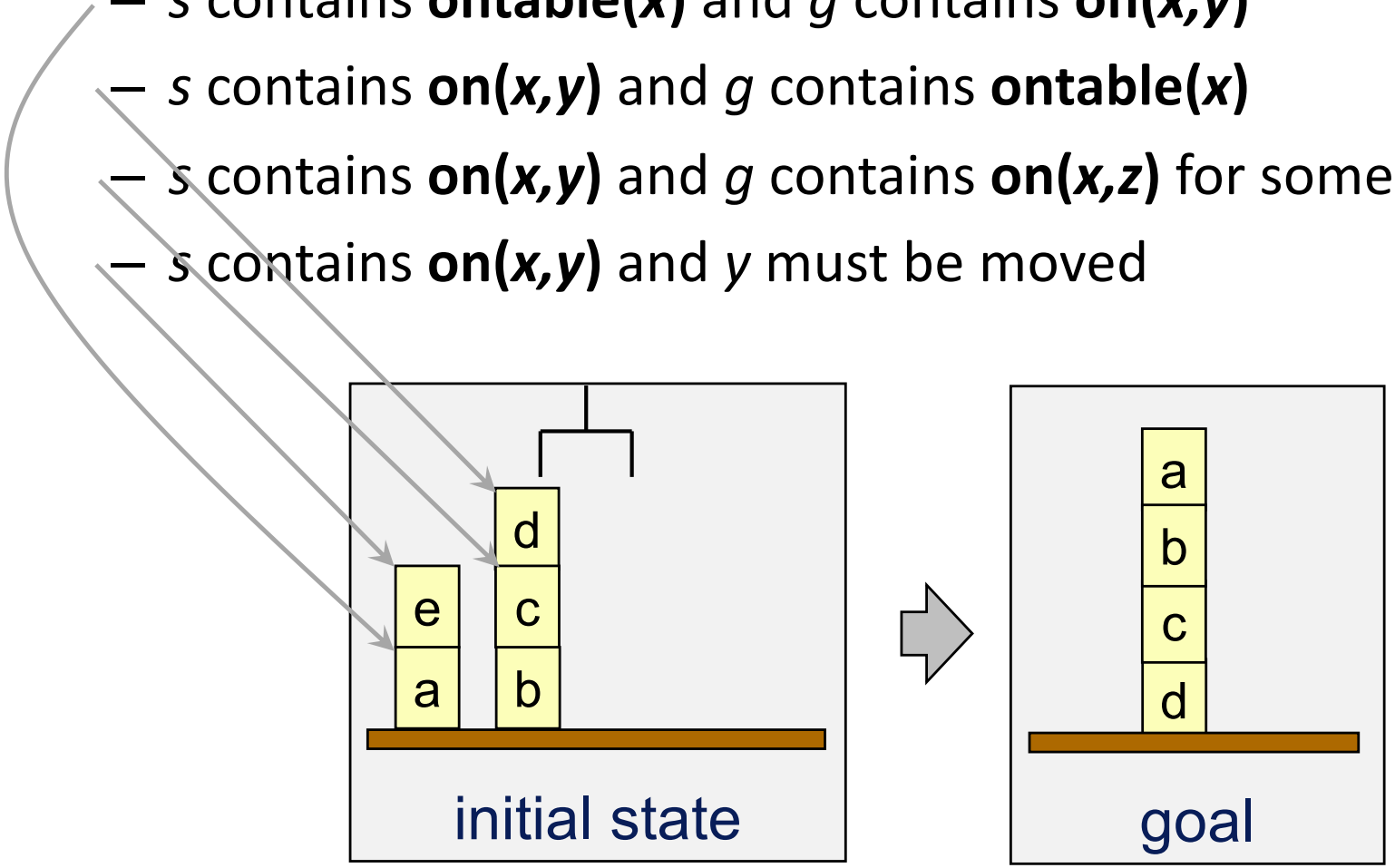
Domain dependent information can prune the search space, but the open question is how to express such information for a general planning algorithm.

- **control rules**

When do we need to move block *x*?

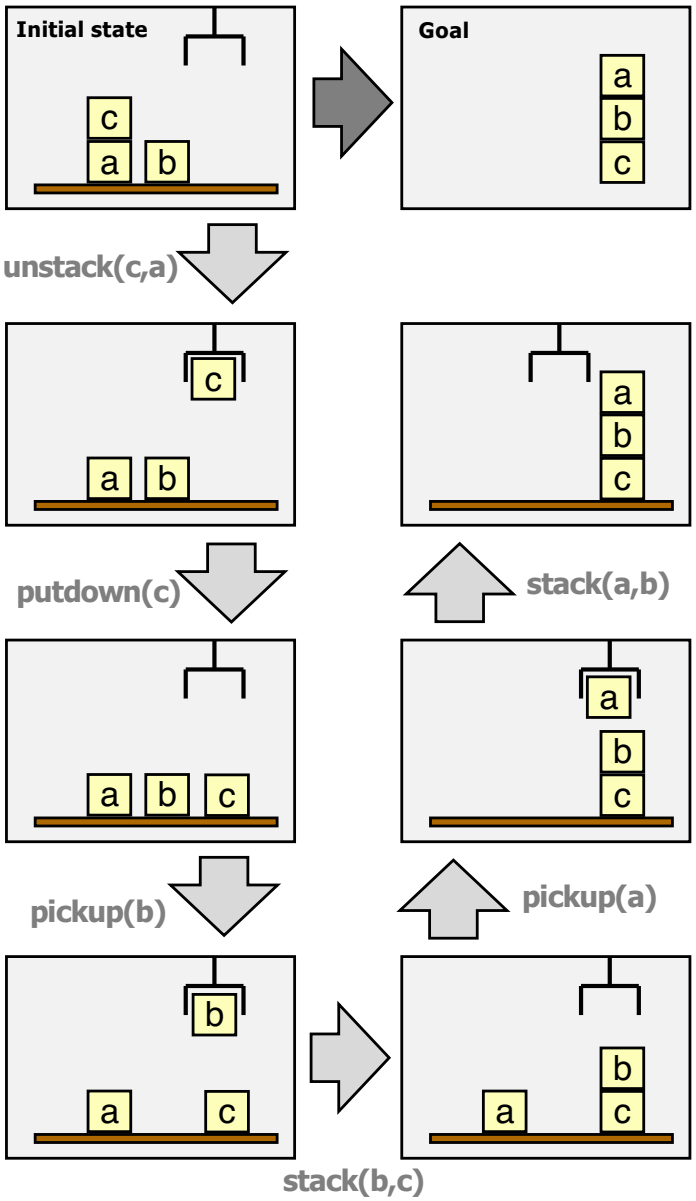
Exactly in one of the following situations:

- *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



Fast planning for blocksworld

```
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*.

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**
  - ϕ_1 U ϕ_2 (until) ϕ_1 is true in all states until the first state (if any) in which ϕ_2 is true
  - $\Box\ \phi$ (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)

Semantics of 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)**:

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

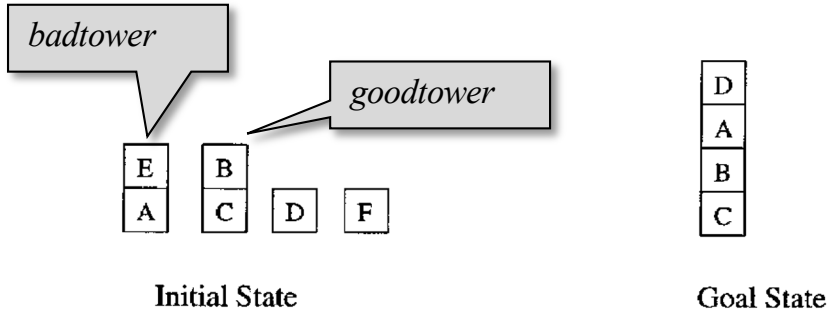
Plan $\pi = \langle a_1, a_2, \ldots, a_n \rangle$ gives a finite sequence of states $S_\pi = \langle s_0, s_1, \ldots, s_n \rangle$ where $s_{i+1} = \gamma(s_i, a_{i+1})$, that can be made infinite $\langle s_0, s_1, \ldots, s_{n-1}, s_n, s_n, s_n, \ldots \rangle$

$(S, s_i, g) \models \phi$ 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 $(S, s_i, g) \models \phi_1$ and $(S, s_i, g) \models \phi_2$
- …
- $(S, s_i, g) \models \phi_1 \cup \phi_2$ iff there exists $j \geq i$ st. $(S, s_j, g) \models \phi_2$ and for each k: $i \leq k < j$ $(S, s_k, g) \models \phi_1$
- $(S, s_i, g) \models \Box \phi$ 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 $(S, s_{i+1}, g) \models \phi$
- $(S, s_i, g) \models \text{GOAL}(\phi)$ 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.



$$goodtower(x) \triangleq clear(x) \wedge \neg \mathrm{GOAL}(holding(x)) \wedge goodtowerbelow(x)$$

$$goodtowerbelow(x) \triangleq (ontable(x) \wedge \neg \exists [y{:}\mathrm{GOAL}(on(x,y))]\,)$$
$$\vee\, \exists [y{:}on(x,y)]\, \neg \mathrm{GOAL}(ontable(x)) \wedge \neg \mathrm{GOAL}(holding(y)) \wedge \neg \mathrm{GOAL}(clear(y))$$
$$\wedge\, \forall [z{:}\mathrm{GOAL}(on(x,z))]\, z = y \wedge \forall [z{:}\mathrm{GOAL}(on(z,y))]\, z = x$$
$$\wedge\, goodtowerbelow(y)$$

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

goodtower remains *goodtower*

Control rule:

$$\Box \big(\forall [x{:}clear(x)]\, goodtower(x) \Rightarrow \bigcirc(clear(x) \vee \exists [y{:}on(y,x)]\, goodtower(y))$$
$$\wedge\, badtower(x) \Rightarrow \bigcirc(\neg \exists [y{:}on(y,x)]\,)$$
$$\wedge\, (ontable(x) \wedge \exists [y{:}\mathrm{GOAL}(on(x,y))]\, \neg goodtower(y))$$
$$\Rightarrow \bigcirc(\neg holding(x))\big)$$

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

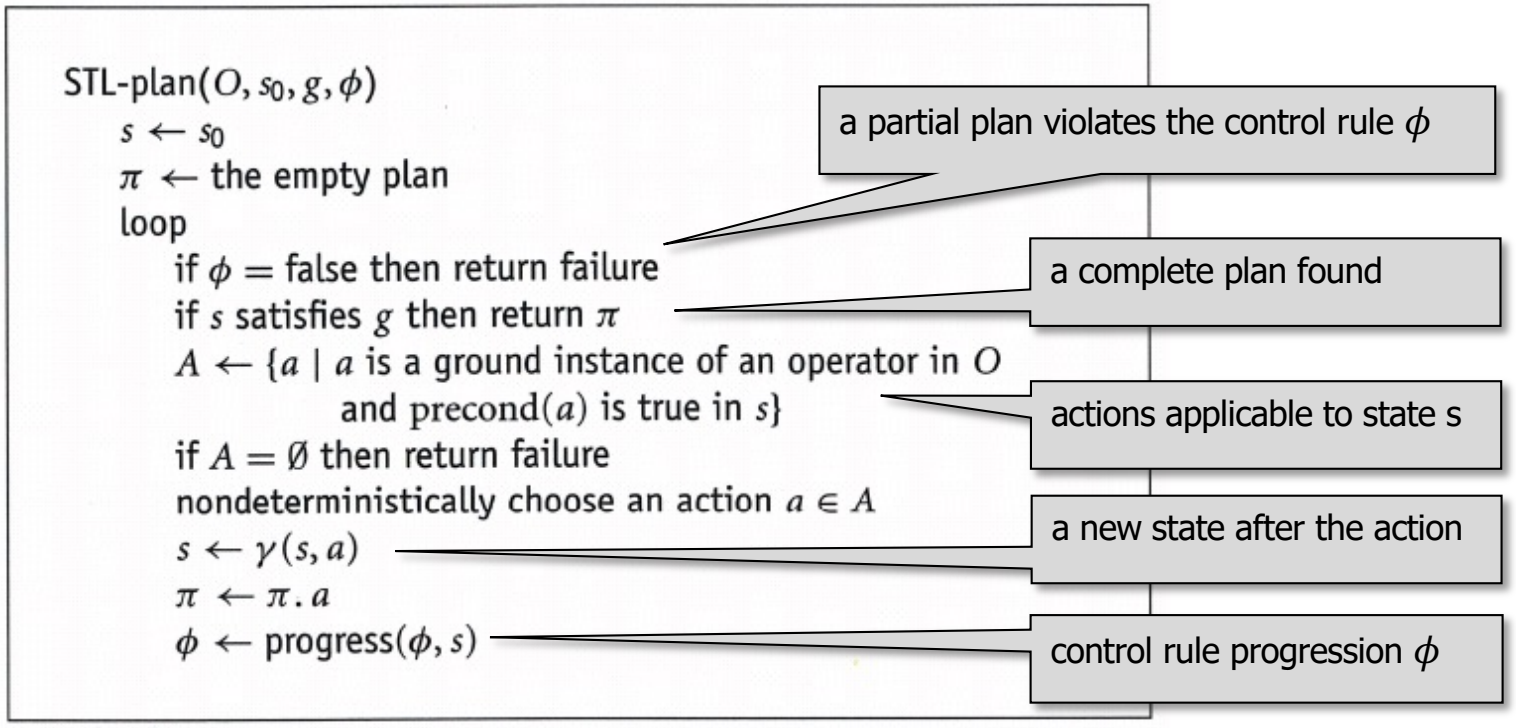
- ϕ does not contain any modal operator
  - progr(ϕ, s_i) = true if $s_i \models \phi$
    = false if $s_i \not\models \phi$ does not hold
- ϕ with logical connectives
  - progr($\phi_1 \wedge \phi_2$, s_i) = progr(ϕ_1, s_i) $\wedge$ progr(ϕ_2, s_i)
  - progr($\neg \phi$, s_i) = $\neg$progr(ϕ, s_i)
-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

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.



Classical planning assumes primitive actions connected via causal relations.

In real-life we can frequently use "**recipes**" to solve a particular task.

- recipe is a set of operations to achieve a sub-goal

HTN planning is based on performing a set of tasks (instead of achieving goals).

- **primitive task**: performed by a classical planning operator
- **non-primitive task**: decomposed by a **method** to other tasks (can use recursion)

How to describe a recipe to perform a given task?

- specify sub-tasks and their relations

A **task network** is a pair (U,C), where U is a set of tasks and C is a set of constraints.

- **tasks** are named similarly to operators: $t(r_1,\ldots,r_n)$
- constraints are in the form:
  - **precedence constraint**: u < v (task u is performed before task v)
  - **before-constraint**: before(U',l) (literal l is true right before the set of tasks U')
  - **after-constraint**: after(U',l) (literal l is true right after the set of tasks U')
  - **between-constraint**: between(U',U'',l) (literal l must be true right after U', right before U'' and in all states in between)

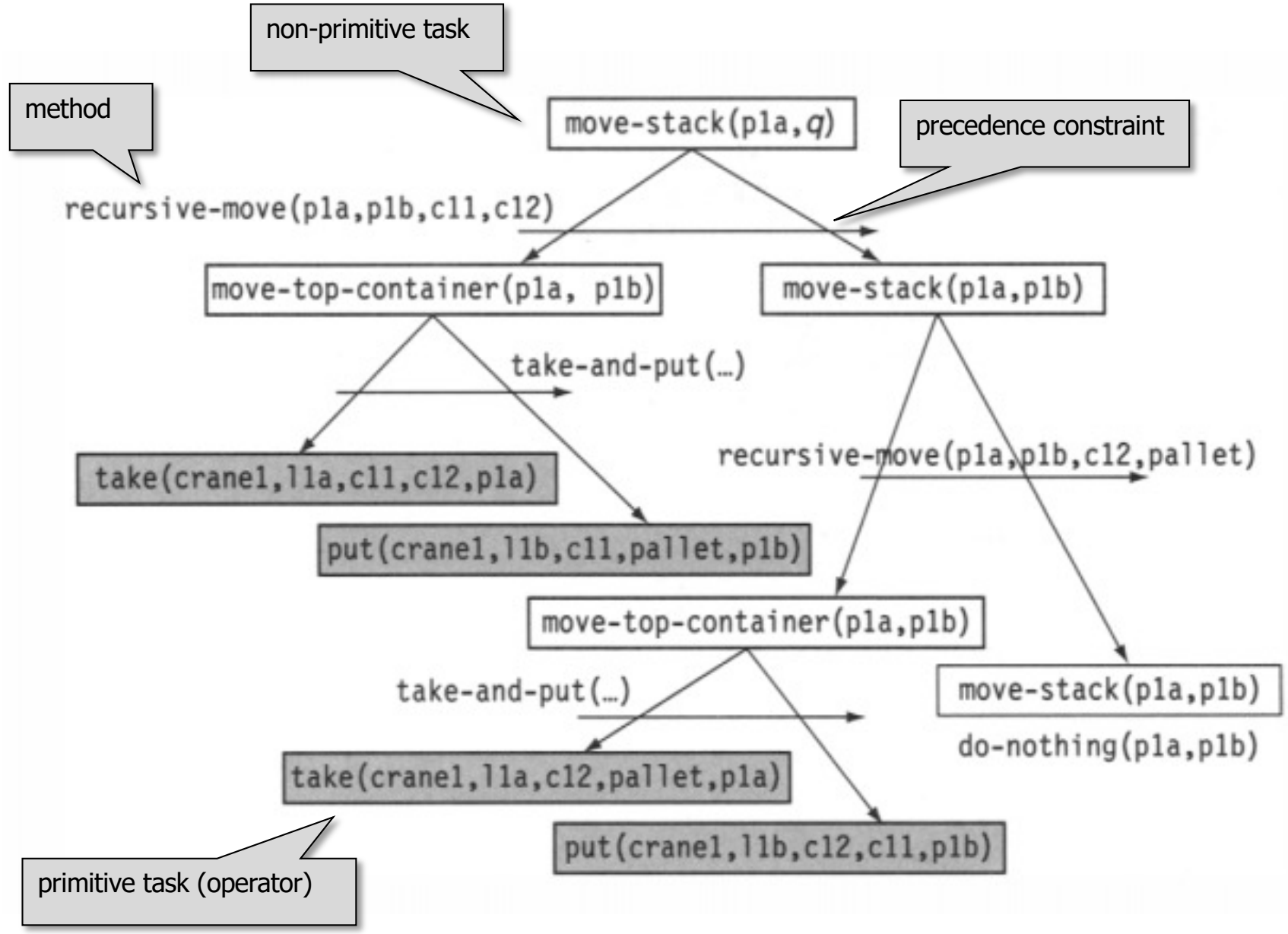
To perform non-primitive tasks, we need to decompose them to other tasks using a method.

An **HTN method** is a tuple
m = (*name, task, subtasks, constr*)

- *name* is $n(x_1,\ldots,x_n)$, where $\{x_1,\ldots,x_n\}$ are all variables in *m* and n is a unique name of the method,
- *task* is a non-primitive task,
- (*subtasks*, *constr*) is a task network.

There may be more methods for a single non-primitive task.

Task decomposition



Now, the planning problem is specified somehow differently from classical planning as a process to obtain a plan from decomposition of tasks in a given task network.

An **HTN planning domain** is a pair (O,M)

- O is a set of operators
- M is a set of HTN methods

An **HTN planning problem** is a 4-tuple (s_0,w,O,M)

- s_0 is the initial state
- w is the initial task network
- (O,M) is the HTN planning domain

When is a plan π a **solution for problem** P?

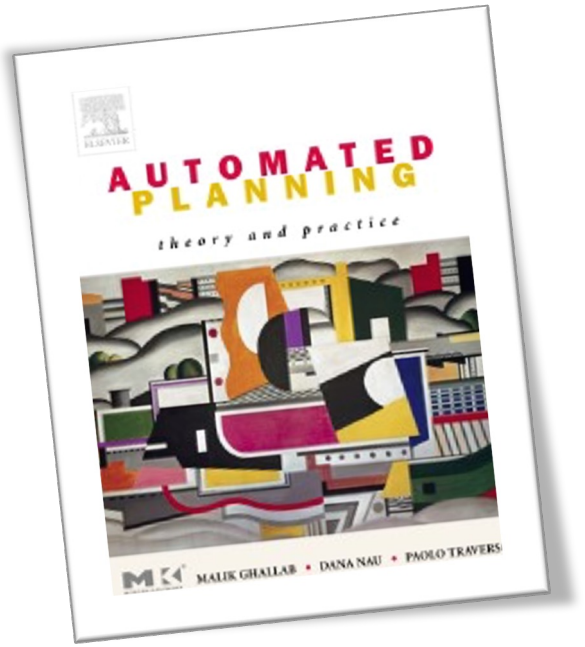
- If w = (U,C) is **primitive** then π = <a_1,…,a_k> is a solution for P, if (U',C') is a ground instance of (U,C) with total ordering <u_1,…,u_k> of nodes in U':
  - the names of tasks <u_1,…,u_k> are actions <a_1,…,a_k>
  - the plan π is executable in the state s_0
  - all constraints C' are satisfied by <a_1,…,a_k>
- If w = (U,C) is **non-primitive** then π is a solution for P if there is a sequence of task decompositions applied to w and giving a primitive task network w' (all tasks are primitive) that is a solution for P.

```
Abstract-HTN(s, U, C, O, M)
    if (U, C) can be shown to have no solution
        then return failure
    else if U is primitive then
        if (U, C) has a solution then
            nondeterministically let π be any such solution
            return π
        else return failure
    else
        choose a nonprimitive task node u ∈ U
        active ← {m ∈ M | task(m) is unifiable with t_u}
        if active ≠ ∅ then
            nondeterministically choose any m ∈ active
            σ ← an mgu for m and t_u that renames all variables of m
            (U', C') ← δ(σ(U, C), σ(u), σ(m))
            (U', C') ← apply-critic(U', C') ;; this line is optional
            return Abstract-HTN(s, U', C', O, M)
        else return failure
```

decomposition of a task

performing application-specific computations

Topics



1. problem formalization
2. classical planning (STRIPS)
   - state-space and plan-space planning
3. neo-classical planning (Graphplan)
   - compilation to SAT and CSP
4. planning with time and resources
   - scheduling task inside planning
5. Heuristics, control knowledge, and hierarchical planning
   - speeding-up planners

Where to learn more?

M. Ghallab, D. Nau, P. Traverso: **Automated Planning: Theory and Practice**, Morgan Kaufmann

Charles University, Prague, Czech Republic
bartak@ktiml.mff.cuni.cz