Automated Planning

Hierarchical Task Networks

Jonas Kvarnström
Automated Planning Group
Department of Computer and Information Science
Linköping University

Some slides adapted from a presentation by Dana Nau

Introduction

- Let's take a **different view** of planning!
  - Instead of a **goal**, let's specify a **task** to perform
    - **goal** at(university) ➔
      task travel-to(university)
  - If I want to **travel-to** some place, I know I can:
    - Walk
    - Go by bike
    - Drive
    - Fly

We can specify alternative **methods**
for performing a **task**

Alternative ➔ must choose which to use ➔ *planning!*

  - If I want to **travel-to** Paris using the **fly** method, I know I have to:

Get a ticket	**Travel** to local airport	**Fly** to remote airport	**Travel** to final destination
	Recursive!		**Recursive!**

We can **decompose** tasks into simpler **subtasks**

Introduction (2)

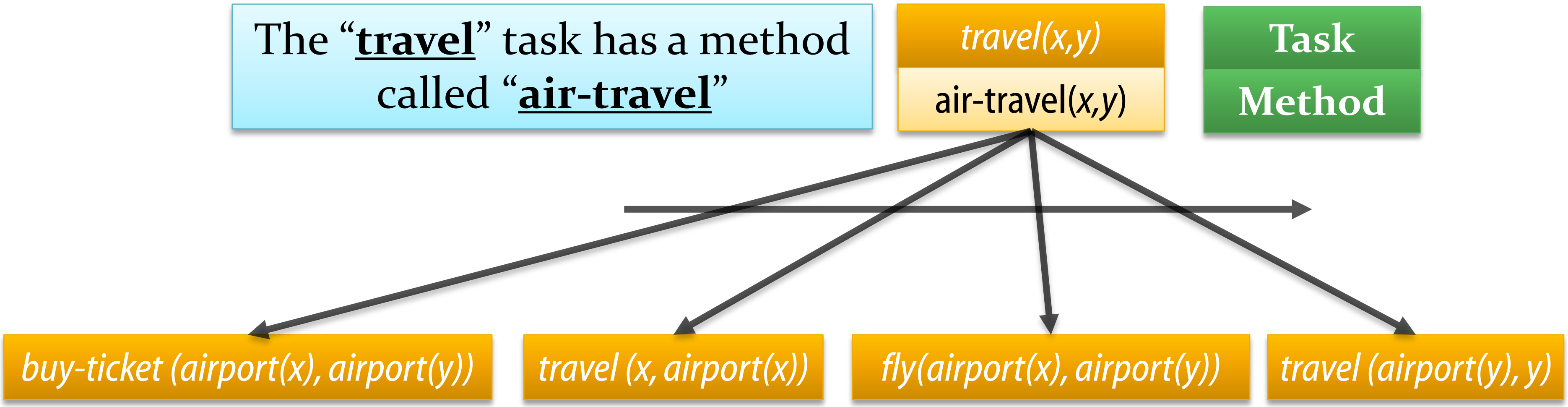
- ➔ **Hierarchical Task Network** planning
  - Instead of goals, we have tasks to perform
  - For each **non-primitive** task:
    - One or more **methods** can be applied, resulting in subtasks
  - A **primitive** task corresponds to an **operator** in standard planning

Total-Order Simple Task Networks

A simple form of Hierarchical Task Network

Totally Ordered STNs

The "**travel**" task has a method called "**air-travel**"

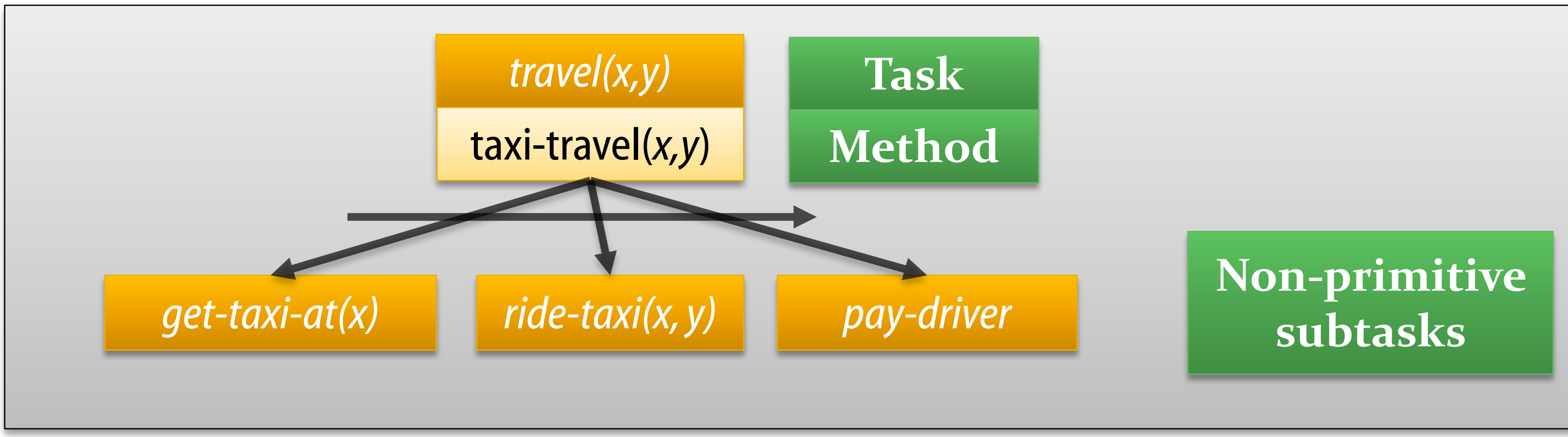


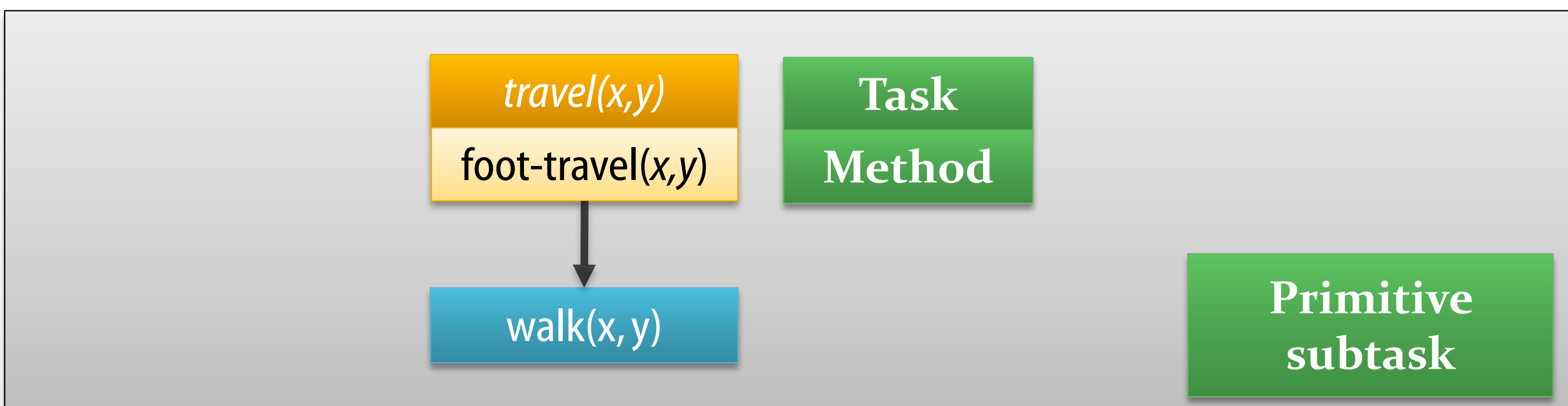
In **Totally Ordered Simple Task Networks (STN),** each method must specify a **sequence** of subtasks (indicated by the horizontal arrow)

Each method can also have a **precondition** (not shown here)

Multiple Methods

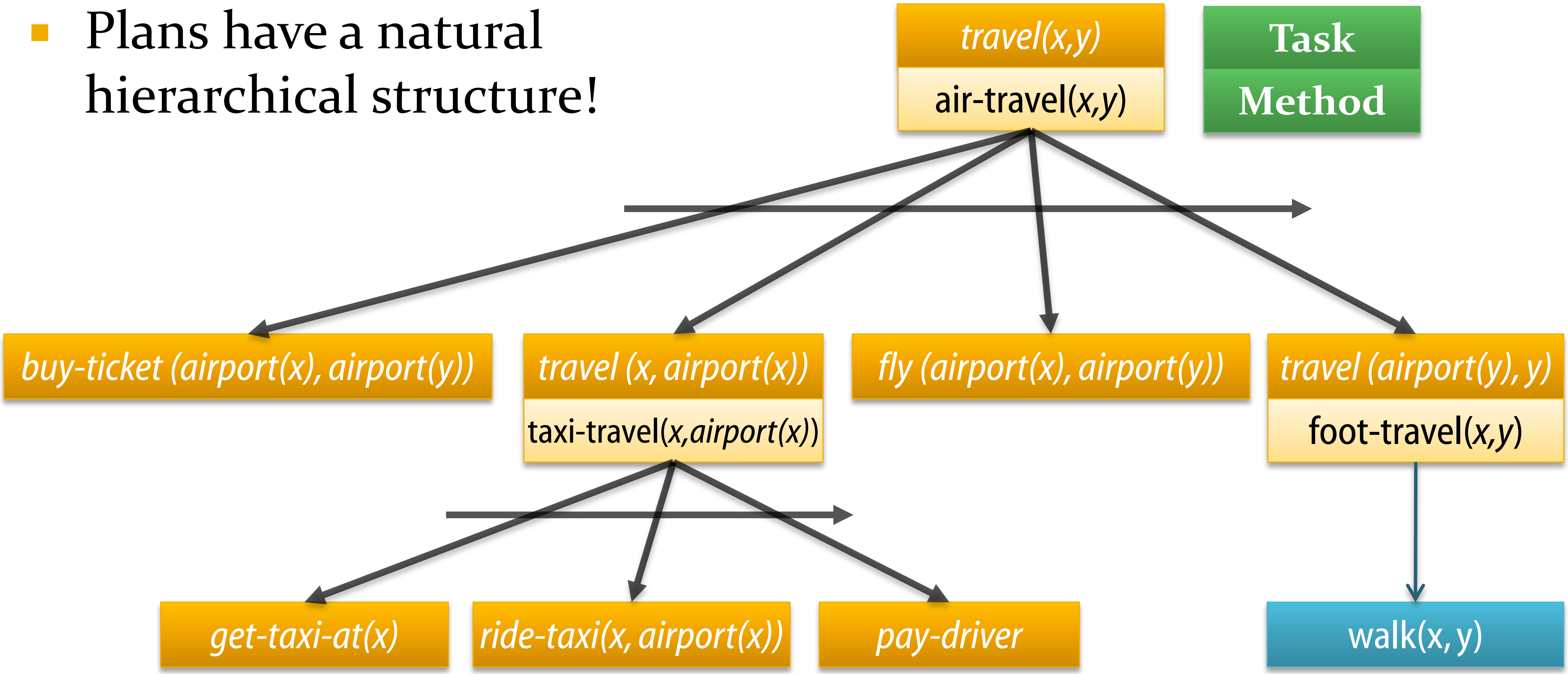
- Any non-primitive task can have **<u>many methods</u>**
  - So you still need to **<u>search</u>**, to determine which method to use
    - You can also **<u>travel</u>** by **<u>taxi-travel</u>** (faster) or **<u>foot-travel</u>** (cheaper)





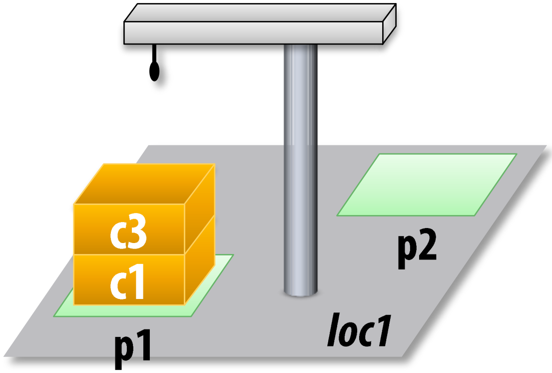
Composition

- Plans have a natural hierarchical structure!



DWR

- Let’s switch to Dock Worker Robots…



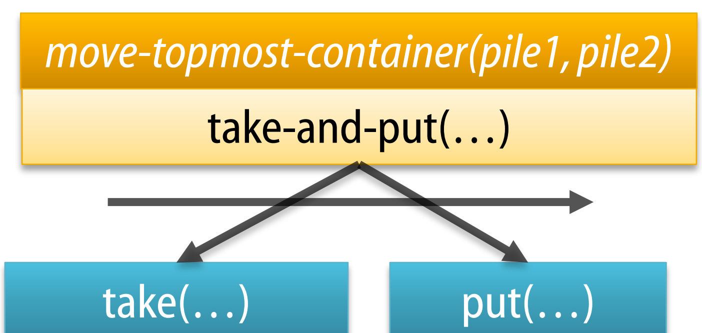
Methods

- To **<u>move the topmost container</u>** from one pile to another:
  - **<u>task</u>**: **move-topmost-container**(pile1, pile2)
  - **<u>method</u>**: **take-and-put**(cont, crane, loc, pile1, pile2, c1, c2)
  - **<u>precond</u>**: attached(pile1, loc), top(cont, pile1), on(cont, c1), attached(pile2, loc), top(c2, pile2), belong(crane, loc)
  - **<u>subtasks</u>**: <**take**(crane, loc, cont, c1, pile1), **put**(crane, loc, cont, c2, pile2)>

In the *task*, we only specify the ”natural” parameters

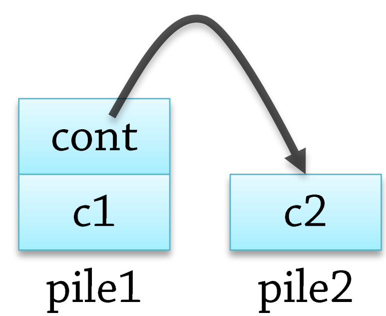
In each *method*, we may use additional parameters whose values are chosen by the planner – just as in classical planning!

Then we use the *precond* to constrain allowed values (cont must be the topmost container of pile1, …)



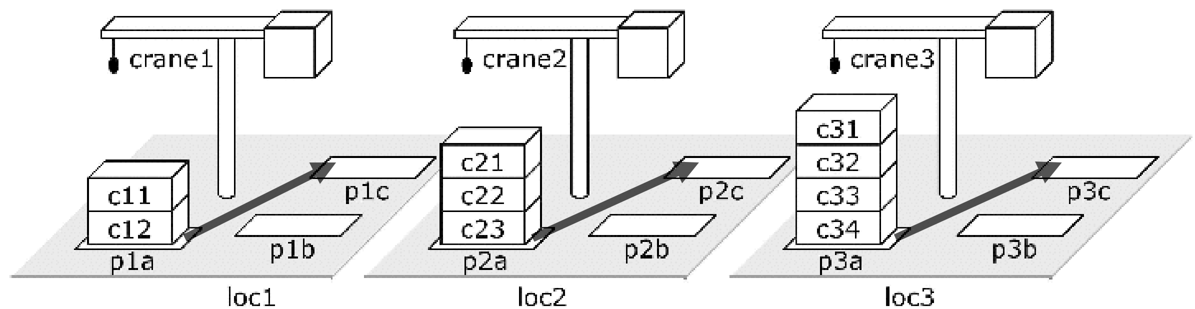
Task

Method

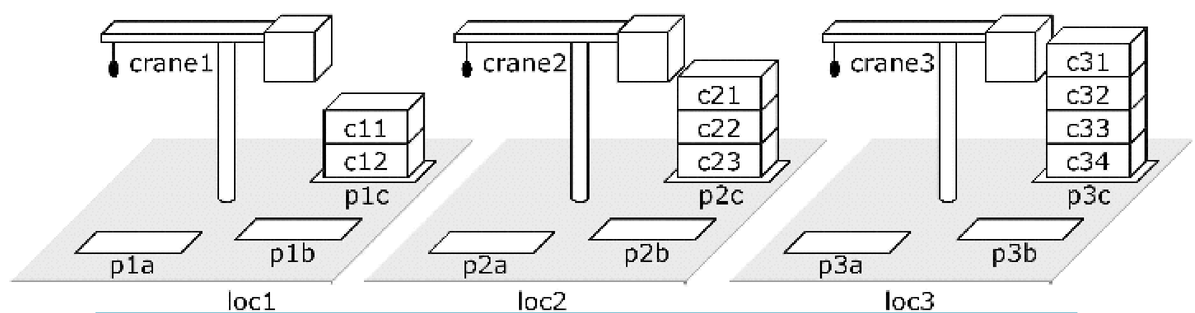


Example

- We want to **move three entire stacks** of containers
  - But **preserve the order** of the containers!
  - Call this **task** move-three-stacks()



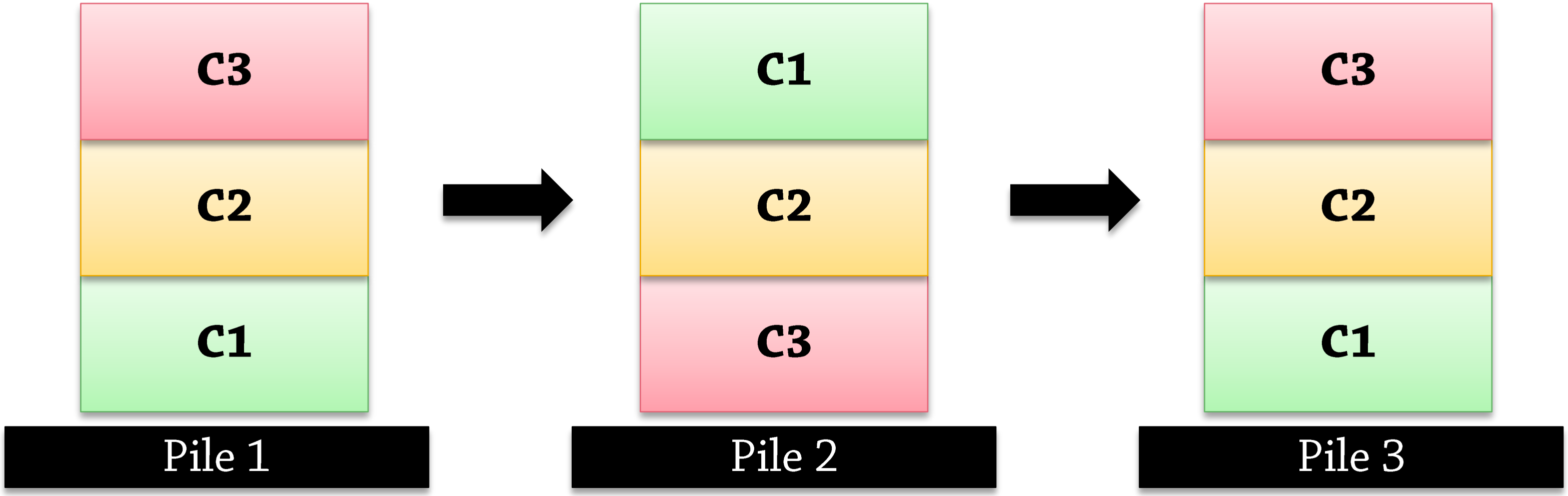
Initial state, with 3 locations, 3 piles to move



Corresponding goal, all piles moved

Example

- How do we do it?
  - First move all containers to **another** pile, so they end up in inverse order
  - Then move them to the real **destination**



Example 2: move-each-twice

- Total-order formulation of move-each-twice:
  - **Task:** move-three-stacks()
    - **method**: move-each-twice()
    - **precond**: ; no preconditions
    - **subtasks**: ; move each stack twice:
      <move-stack(p1a, p1b), move-stack(p1b, p1c),
      move-stack(p2a, p2b), move-stack(p2b, p2c),
      move-stack(p3a, p3b), move-stack(p3b, p3c) >

All subtasks are sequentially ordered

Example 2b: move-each-twice

- Alternative total-order formulation of move-each-twice:
  - <u>**Task:**</u> move-three-stacks()
    - **method**: move-each-twice(**loc1, interm1, loc2, interm2, loc3, interm3, …**)
    - **precond**: top(pallet, interm1), top(pallet, interm2), top(pallet, interm3), attached(…), …

      Let the planner choose an intermediate pile (there might be several alternatives)!

    - **subtasks**: ; move each stack twice:
      <move-stack(p1a, **interm1**), move-stack(**interm1**, p1c),
      move-stack(p2a, **interm2**), move-stack(**interm2**, p2c),
      move-stack(p3a, **interm3**), move-stack(**interm3**, p3c) >

Supporting Methods and Tasks

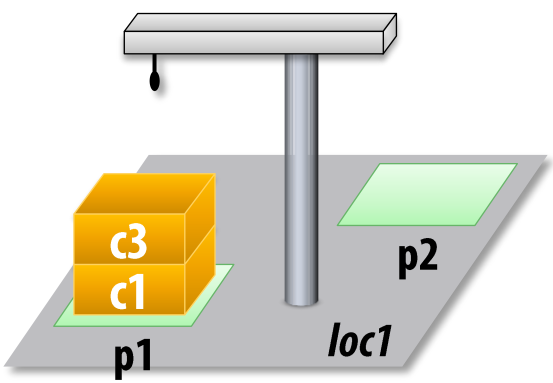
- How can we implement the task **move-stack**(pile1, pile2)?
  - Must move **<u>all</u>** containers in a stack, but we *don't know how many…*
  - HTN planning allows **<u>recursion</u>**
    - Move the **<u>topmost</u>** container (we know how to do that!)
    - Then move the **<u>rest</u>**
  - First attempt: **<u>Task</u>** move-stack(pile1, pile2)
    - **method**: recursive-move(pile1, pile2)
    - **precond**: true
    - **subtasks**: <move-topmost-container(pile1, pile2), move-stack(pile1, pile2)>

But the bottom of the pile is the pallet, and we don't want to move that!

In the BW, we had an "ontable" predicate.
The bottom block was not "on" another block.

In DWR: A special "bottom object" in each pile, the **<u>pallet</u>**.

top(c3, p1) – on(c3, c1) – on(c1, pallet)



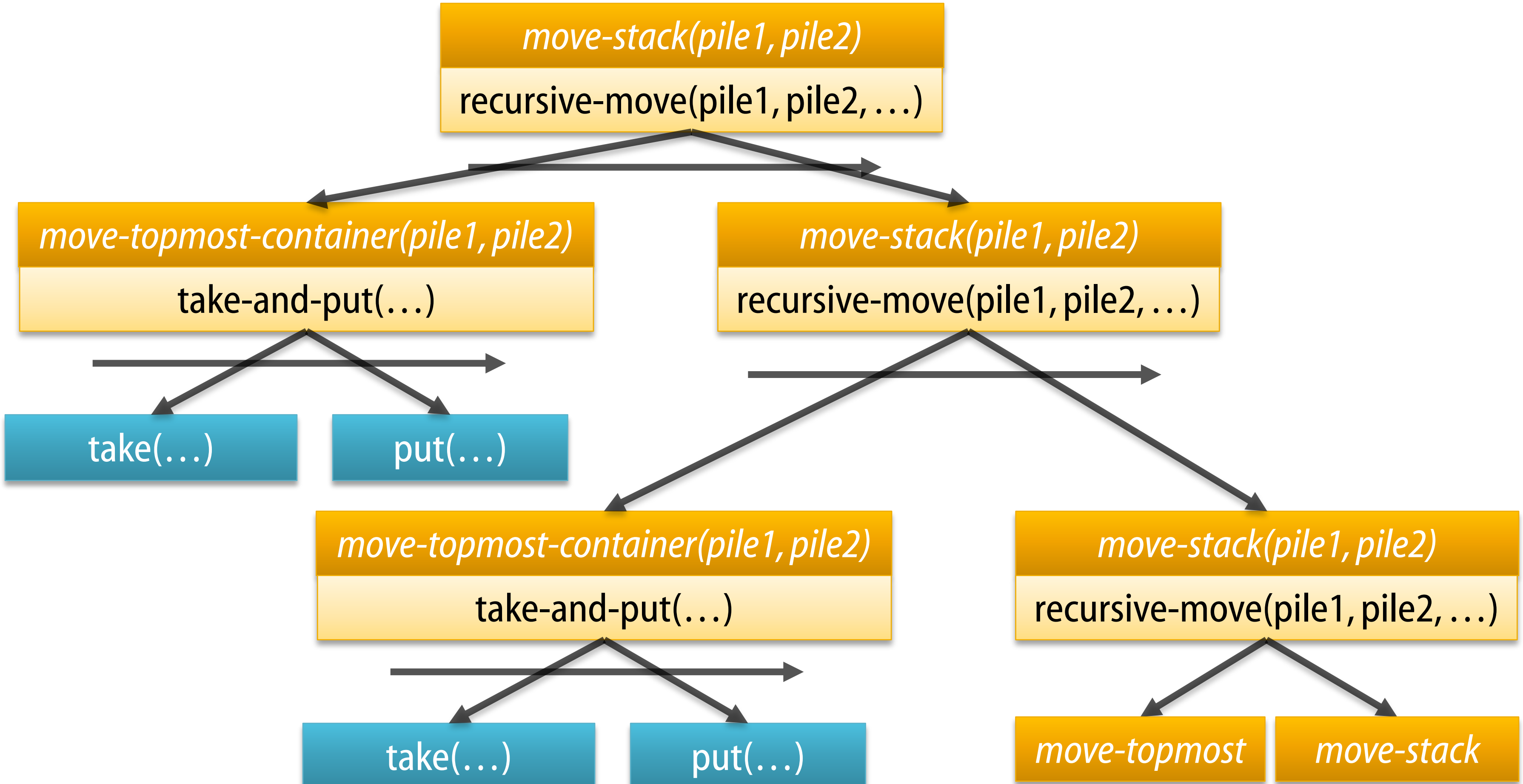
Supporting Methods and Tasks (2)

- Problem fixed: **<u>Task</u>** move-stack(pile1, pile2)
  - Method recursive-move(pile1, pile2, *cont*, *x*)
  - precond: <u>top(*cont*, pile1), on(*cont*, *x*)</u>
  - subtasks: <move-topmost-container(pile1, pile2), move-stack(pile1, pile2)>

The topmost container is on top of something (x), so it can't be the pallet

Recursion

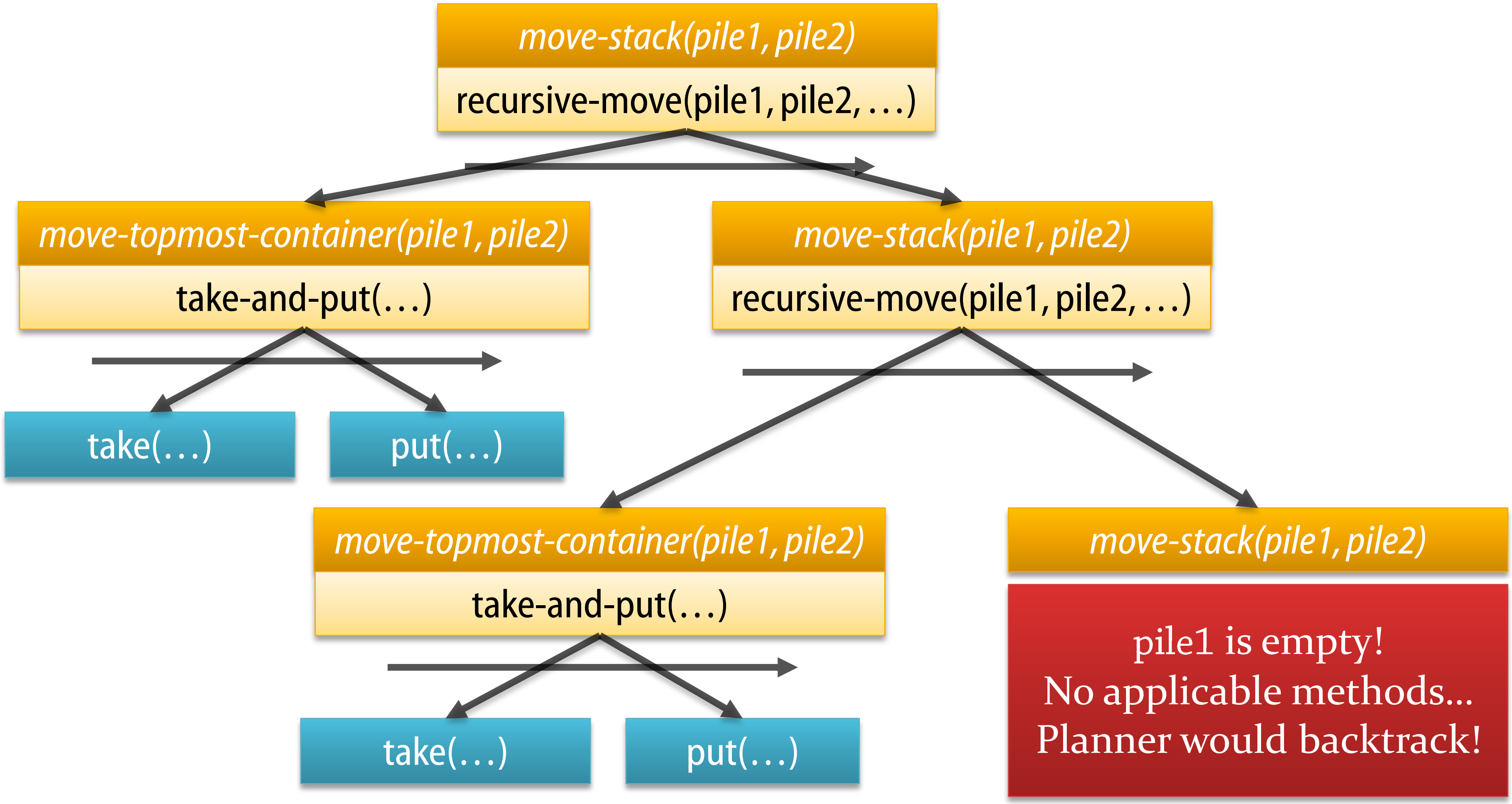
- The planner can now create a structure like this:



But when will the recursion end?

Supporting Methods and Tasks

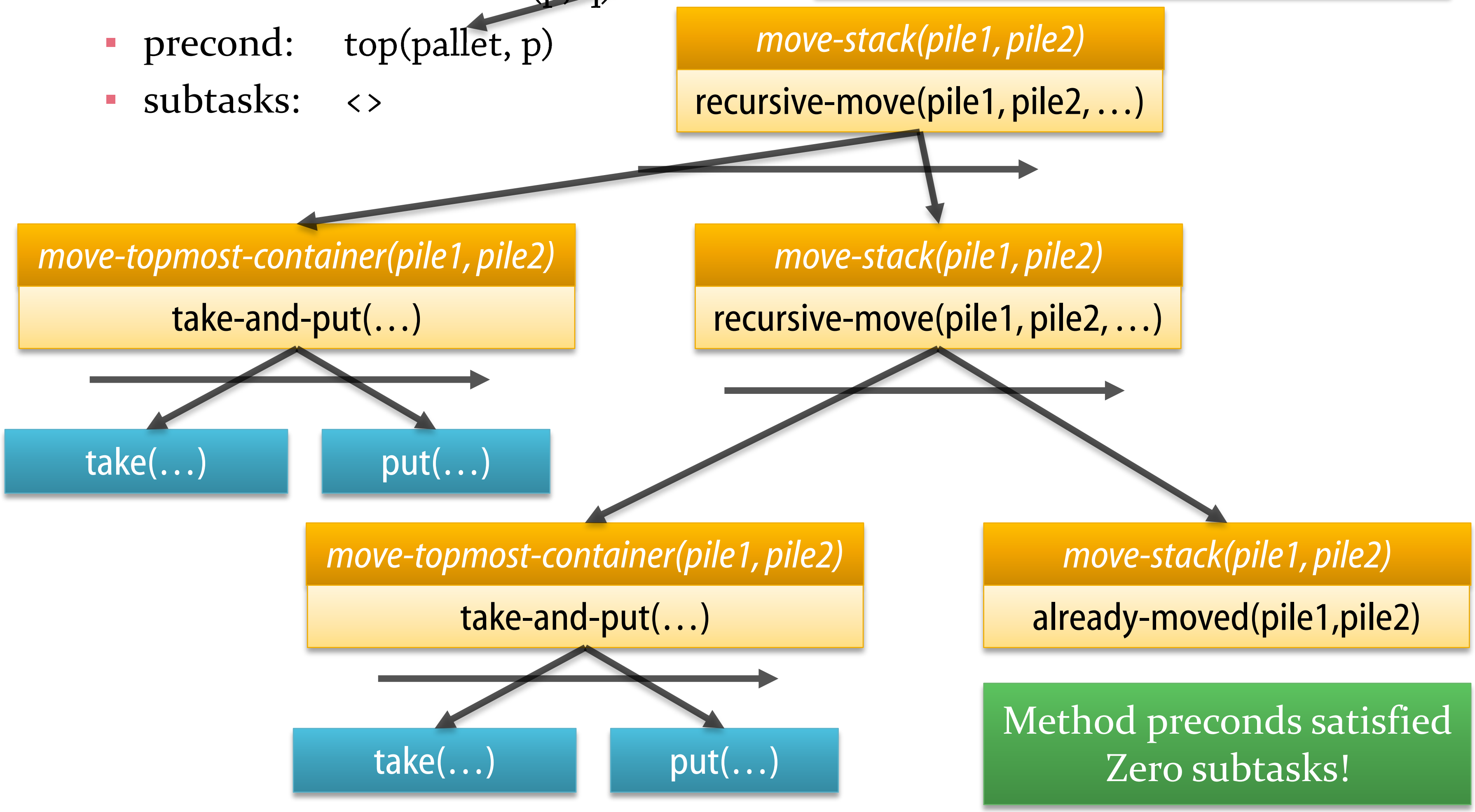
- At some point, only the pallet will be left in the stack
  - Then recursive-move will not be applicable
  - But we specified that we **must** execute **some** form of move-stack!



Supporting Methods and Tasks

- We must have a method that can **terminate** the recursion
  - **Method** already-moved(p, q)
    - task: move-stack(p, q)
    - precond: top(pallet, p)
    - subtasks: <>

Unique pallet object –
not a variable!



Domains, Problems, Solutions

- An HTN **planning domain** specifies:
  - **Tasks** that are available (primitive and non-primitive)
  - **Methods** to decompose non-primitive tasks into subtasks
  - **Constraints** to be enforced
    - E.g., don't use a taxi for long distances
- An HTN **problem instance** specifies:
  - **Initial state** information
  - One or more **tasks** to perform, with concrete parameters
    - For Total Order Simple Task Networks:
A *sequence* of tasks to perform

No goals to be achieved!
We should **perform tasks**.

Domains, Problems, Solutions

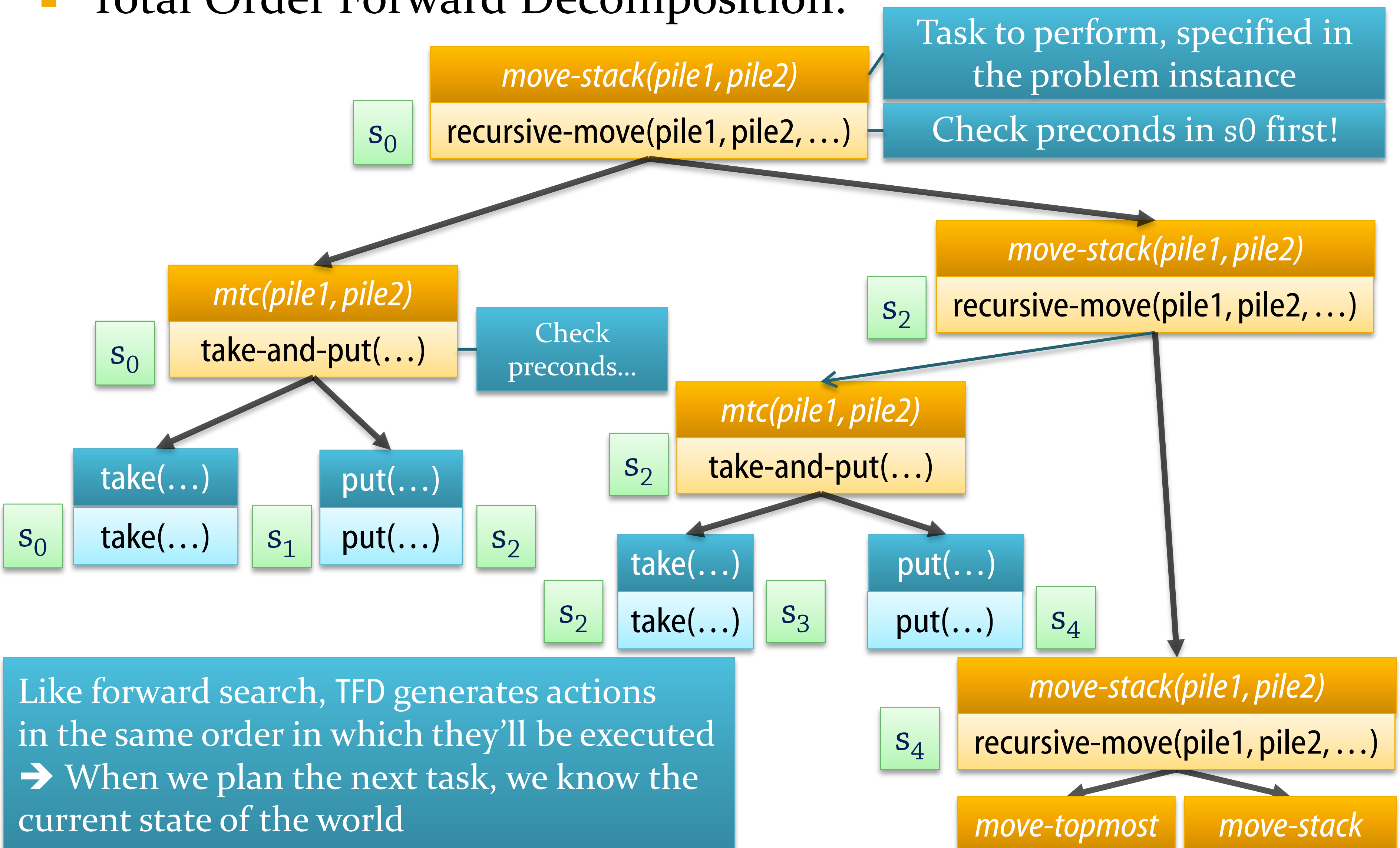
- A **solution** is any **executable action sequence** that can be generated from the initial task(s) by recursively applying
  - **methods** to non-primitive tasks
  - **operators** to primitive tasks
  - (No goals to be achieved)

- The planner uses **only** the methods specified for a given task
  - Will **not** try arbitrary actions…
  - For this to be useful, you must **have** useful "recipes" for all tasks

A Planning Algorithm: Total Order Forward Decomposition

Total Order Forward Decomposition

- Total Order Forward Decomposition:



Like forward search, TFD generates actions in the same order in which they'll be executed
➔ When we plan the next task, we know the current state of the world

Solving Total-Order STN Problems (1)

- TFD takes four inputs:
  - s – the current state
  - <t1,…,tk> – a list of tasks to be achieved **in the specified order**
  - O – the available operators
  - M – the available methods

  - TFD(s, <t1,…,tk>, O, M):
    - // If we have no tasks left to do…
      if (k = 0) **then** return the empty plan

Solving Total-Order STN Problems (2)

If tasks are ground...

- TFD(s, <t1,…,tk>, O, M):
  - if (k = 0) then return the empty plan
  - **<u>if</u>** (t1 is primitive) **<u>then</u>**

    // A primitive task is decomposed into a single action!
    // There may be many to choose from.
    actions ← ground instances of operators in O
    candidates ← { a | a ∈ *actions* and
    a is relevant for t1 and // Achieves the task
    a is applicable in s }
    <u>if</u> (*candidates* = ∅) return failure

  - **<u>nondeterministically choose</u>** any a ∈ *candidates* // Or use backtracking
  - *newstate* ← γ(s,a) // Apply the action, find the new state

    remaining ← <t2,…,tk>
    π ← *TFD(newstate, remaining*, O, M)
    <u>if</u> (π = failure) return failure
    <u>else</u> return a.π // Concatenation: a + the rest of the plan

Solving Total-Order STN Problems (3)

- TFD(s, <t1,…,tk>, O, M):
  - if (k = 0) then return the empty plan
  - **<u>if</u>** (t1 is primitive) **<u>then</u>**

    // A primitive task is decomposed into a single action!
    // May be many to choose from (e.g. method has more params than task).
    actions ← ground instances of operators in O
    candidates ← { (a,**σ**) | a ∈ *actions* and
    σ **is a substitution** s.t. action *a* **achieves σ(t1)** and
    a is applicable in s }
    <u>if</u> (*candidates* = ∅) return failure

  - **<u>nondeterministically choose</u>** any **(a,σ)** ∈ *candidates*// Or use BT
  - *newstate* ← γ(s,a) // Apply the action, find the new state
    remaining ← **σ(<t2,…,tk>)** // Must have the same variable bindings!
    π ← *TFD(newstate, remaining*, O, M) // Handle the remaining tasks
    <u>if</u> (π = failure) return failure
    <u>else</u> return a.π

If tasks can be **<u>non-ground</u>**:
move(container1,*X*)

Basically, σ can specify variable bindings for parameters of t1…

Solving Total-Order STN Problems

- TFD(s, <t1,…,tk>, O, M):
  - if (k = 0) then return the empty plan
  - **if** (t1 is primitive) **then** …
  - **else** // t1 is travel(LiU, Resecentrum), for example

    // A non-primitive task is decomposed into a new task list.

    // May have many methods to choose from: taxi-travel, bus-travel, walk, …

    ground ← ground instances of methods in M

    candidates ← { (m,σ) | m ∈ *ground* and σ is a substitution s.t. task(m) = σ(t1) and m is applicable in s } // Methods have preconds!

    if (*candidates* = ∅) return failure

    nondeterministically choose any (m,σ) ∈ active // Or use backtracking

    // No actions are applied here!

    remaining ← **subtasks(m)** . σ(<t2,…,tk>) // Prepend new list!

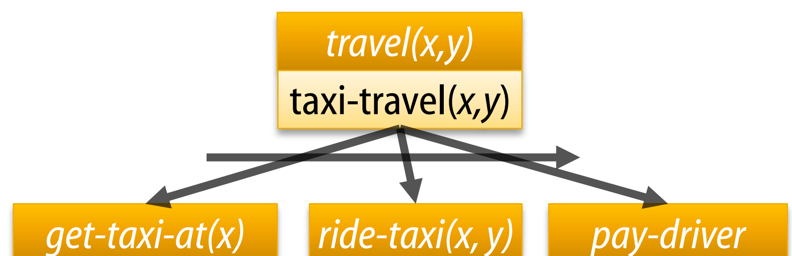
    π ← *TFD(s, remaining*, O, M)

    if (π = failure) return failure

    else return π

As before, but methods instead of actions

Replace the task by its subtasks



Limitations of Total-Order HTN Planning

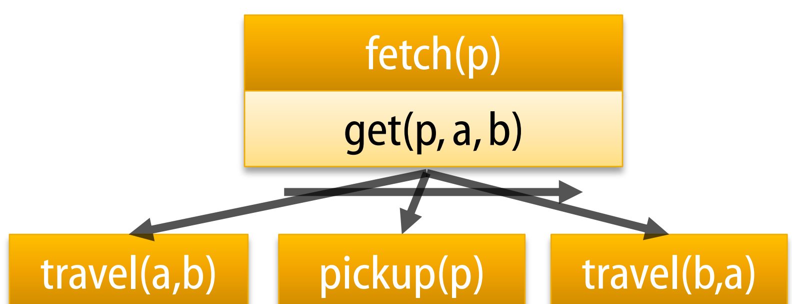
Limitation of Ordered-Task Planning

- TFD requires **totally ordered** methods
  - Can't interleave subtasks of different tasks
- Suppose we want to **fetch one object** somewhere, then return to where we are now

  - Task: **fetch**(obj)
    - method: **get**(obj, mypos, objpos)
      - precond: **robotat**(mypos) & at(obj, objpos)
      - subtasks: <**travel**(mypos, objpos), **pickup**(obj), **travel**(objpos, mypos)>

  - Task: **travel**(x, y)
    - method: **walk**(x, y)
    - method: **stayat**(x)

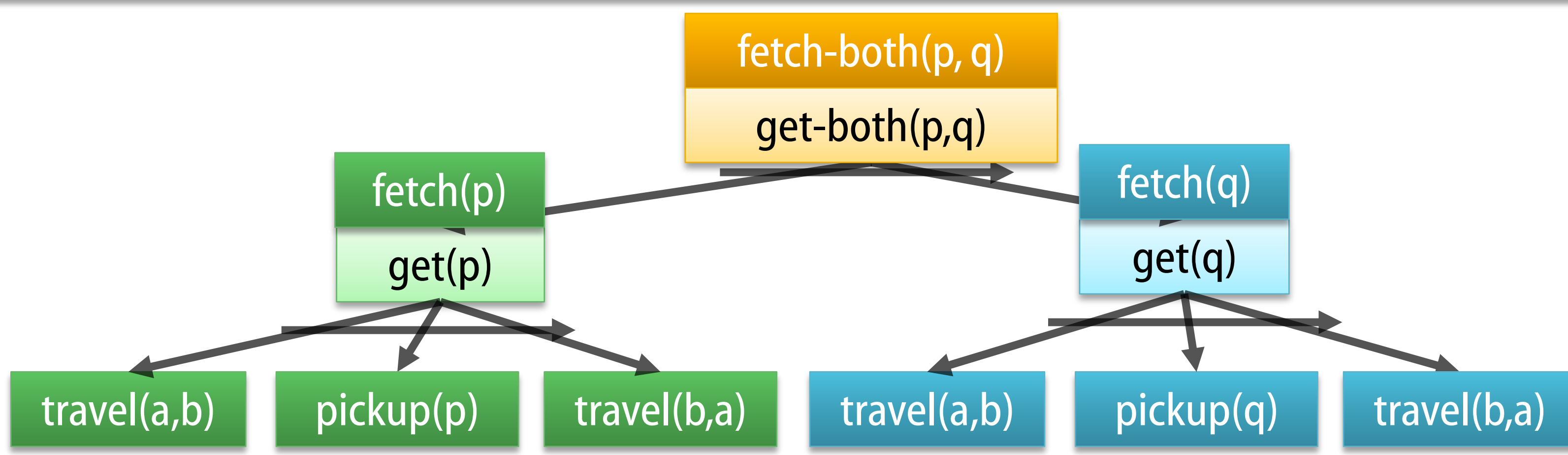
I'm at A, the thing to fetch is at B



Limitation of Ordered-Task Planning

- Suppose we want to fetch **two** objects somewhere, and return
  - (Simplified example – consider "fetching all the objects we need")
- One idea: Just "fetch" each object in sequence
  - Task: **fetch-both**(obj1, obj2)
    - method: **get-both**(obj1, obj2, mypos, objpos1, objpos2)
      - precond: –
      - subtasks: <**fetch**(obj1, mypos, objpos1), **fetch**(obj2, mypos, objpos2)>

I'm at A, both objects are at B



Have to start with the first Fetch…

I'm back at A and have to walk again!

Alternative Methods

- To generate more efficient plans using total-order STNs:
  - Use a different domain model!

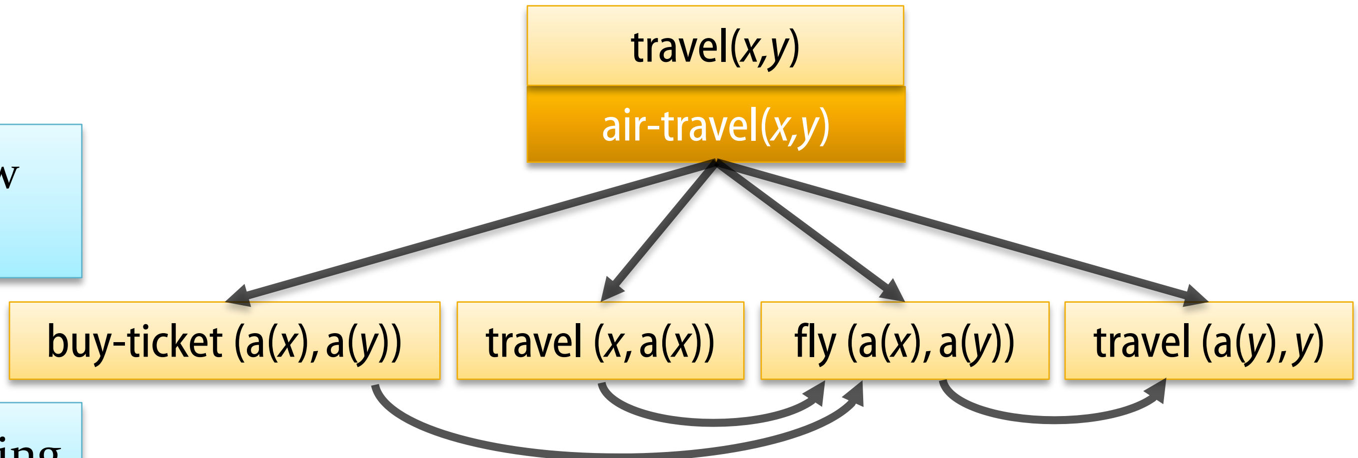
  - Task: **fetch-both**(obj1, obj2)
    - method: **get-both**(obj1, obj2, mypos, objpos1, objpos2)
      - precond: objpos1 != objpos2 & at(obj1, objpos1) & at(obj2, objpos2)
      - subtasks: <**travel**(mypos, objpos1), **pickup**(obj1), **travel**(objpos1, objpos2), **pickup**(obj2), **travel**(objpos2, mypos)>

  - Task: **fetch-both**(obj1, obj2)
    - method: **get-both-in-same-place**(obj1, obj2, mypos, objpos)
      - precond: **robotat**(mypos) & at(obj1, objpos) & at(obj2, objpos)
      - subtasks: <**travel**(mypos, objpos), **pickup**(obj1), **pickup**(obj2), **travel**(objpos, mypos)>

HTN Planning with Partially Ordered Methods

Partially Ordered Methods

- **<u>Partially ordered method</u>**:
  - The subtasks are a **<u>partially ordered</u>** set $\{t_1, \ldots, t_k\}$



No horizontal arrow ordering all tasks

Indicate partial ordering

method air-travel(x,y)

task: travel(x,y)

precond: long-distance(x,y)

network: u_1=buy-ticket($a(x)$,$a(y)$), u_2= travel(x,$a(x)$), u_3= fly($a(x)$, $a(y)$)

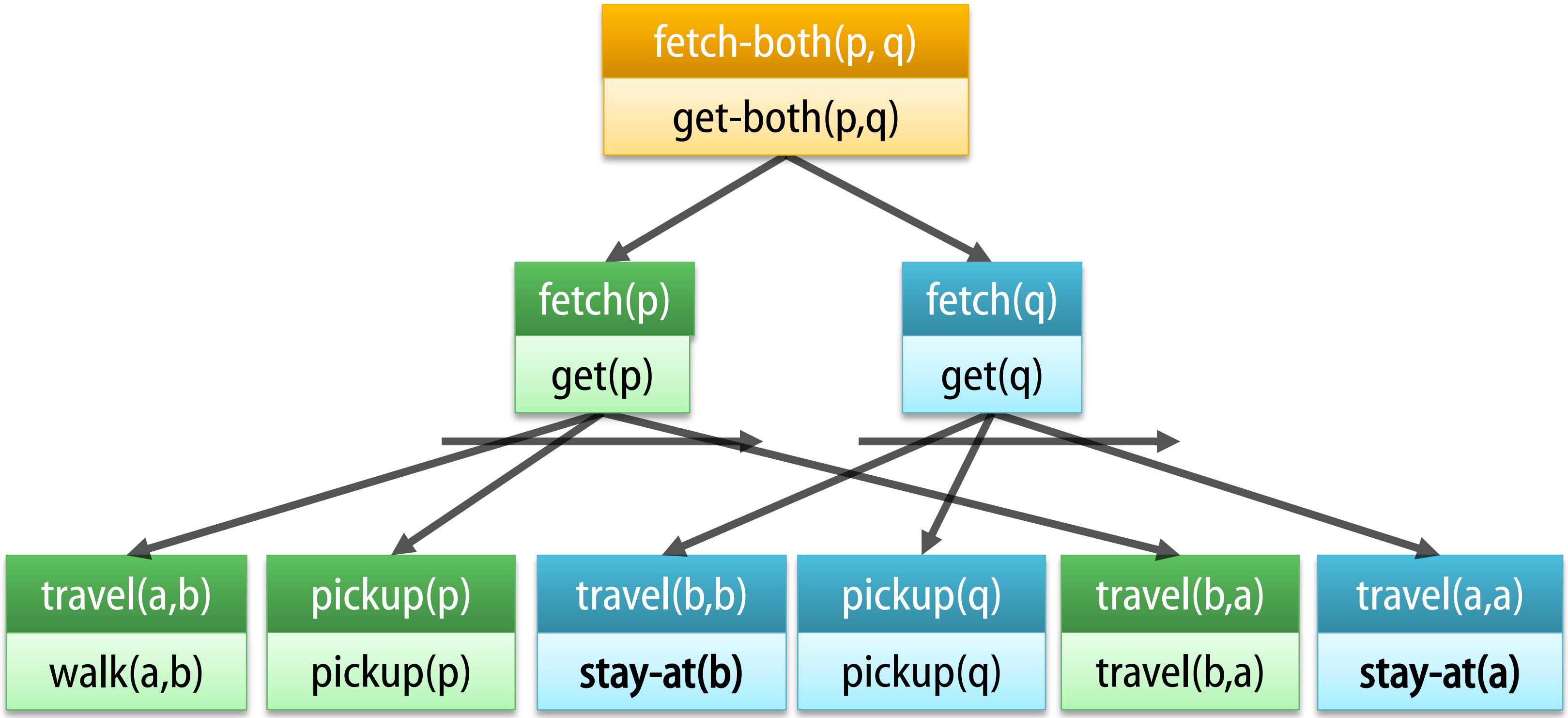
u_4= travel($a(y)$,y),

$\{(u_1,u_3), (u_2,u_3), (u_3,u_4)\}$

Precedence: u1 before u3, etc.

Partially Ordered Methods

- With partially ordered methods, **subtasks can be interleaved**



  - Requires a more complicated planning algorithm: PFD
  - SHOP2: implementation of PFD-like algorithm + generalizations

Partial-Order and Total-Order

- Partial-order formulation of move-each-twice:

move-each-twice()
  task: move-all-stacks()
  precond: *; no preconditions*
  network: *; move each stack twice:*
  u_1 =move-stack(p1a,p1b), u_2 =move-stack(p1b,p1c),
  u_3 =move-stack(p2a,p2b), u_4 =move-stack(p2b,p2c),
  u_5 =move-stack(p3a,p3b), u_6 =move-stack(p3b,p3c),
  $\{(u_1, u_2), (u_3, u_4), (u_5, u_6)\}$

Each stack is moved to the temp pile before it is moved to its final pile
Otherwise, no ordering constraints

  - Old total-order formulation:

move-each-twice()
  task: move-all-stacks()
  precond: *; no preconditions*
  subtasks: *; move each stack twice:*
  ⟨move-stack(p1a,p1b), move-stack(p1b,p1c),
  move-stack(p2a,p2b), move-stack(p2b,p2c),
  move-stack(p3a,p3b), move-stack(p3b,p3c)⟩

Solving Partial-Order STN Problems (1)

- PFD takes four inputs:
  - s – the current state
  - w – a **<u>network/graph</u>** of tasks to be achieved
  - O – the available operators
  - M – the available methods

- PFD(s, w, O, M):
  - // If we have no tasks left to do…
    <u>if</u> (w = emptyset) **<u>then</u>** return the empty plan

Solving Partial-Order STN Problems (2)

- TFD(s, <t1,…,tk>, O, M):
  - if (w = emptyset) then return the empty plan
  - **nondeterministically choose** a task u in w that has no predecessors
  - **if** (u is primitive) **then**

    actions ← ground instances of operators in O
    candidates ← { (a,**σ**) | a ∈ *actions* and
    σ **is a substitution** s.t. action *a* **achieves σ(t1)** and
    a is applicable in s }
    if (*candidates* = ∅) return failure

    nondeterministically choose any **(a,σ)** ∈ *candidates*

  - *newstate* ← γ(s,a) // Apply the action, find the new state
    remaining ← **σ(w – {u})** // Must have the same variable bindings!
    π ← *PFD*(*newstate*, *remaining*, O, M) // Handle the remaining tasks
    if (π = failure) return failure
    else return a.π

A task that **can be first** – not necessarily a unique "first task"!

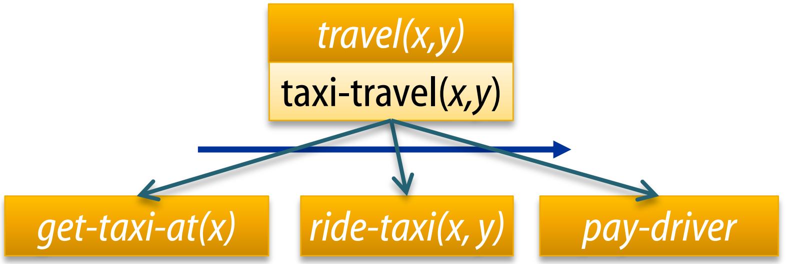
Solving Partial-Order STN Problems (3)

- TFD(s, <t1,…,tk>, O, M):
  - if (w = emptyset) then return the empty plan
  - **if** (u is primitive) **then** …
  - **else** // u is travel(LiU, Resecentrum), for example

    ground ← ground instances of methods in M
    candidates ← { (m,σ) | m ∈ *ground* and
    σ is a substitution s.t. task(m) = σ(t1) and
    m is applicable in s } // Methods have preconds!
    if (*candidates* = ∅) return failure
    nondeterministically choose any (m,σ) ∈ active // Or use backtracking

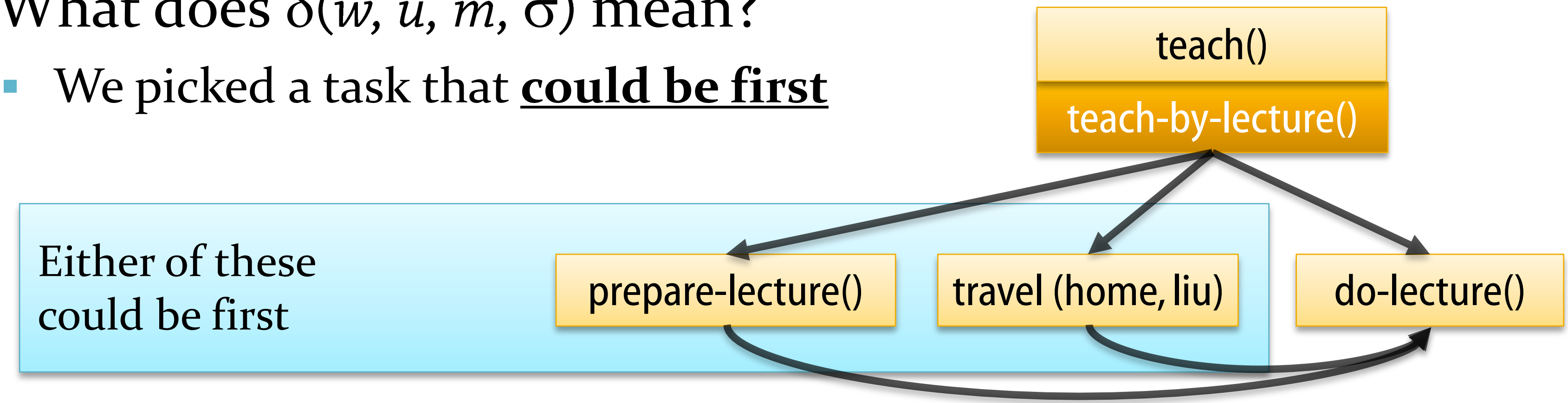
  - // No actions are applied here!
    remaining ← $\delta(w, u, m, \sigma)$
    π ← *PFD(s, remaining,* O, M)
    if (π = failure) return failure
    else return π

Replacing the task by its subtasks is more complicated here!

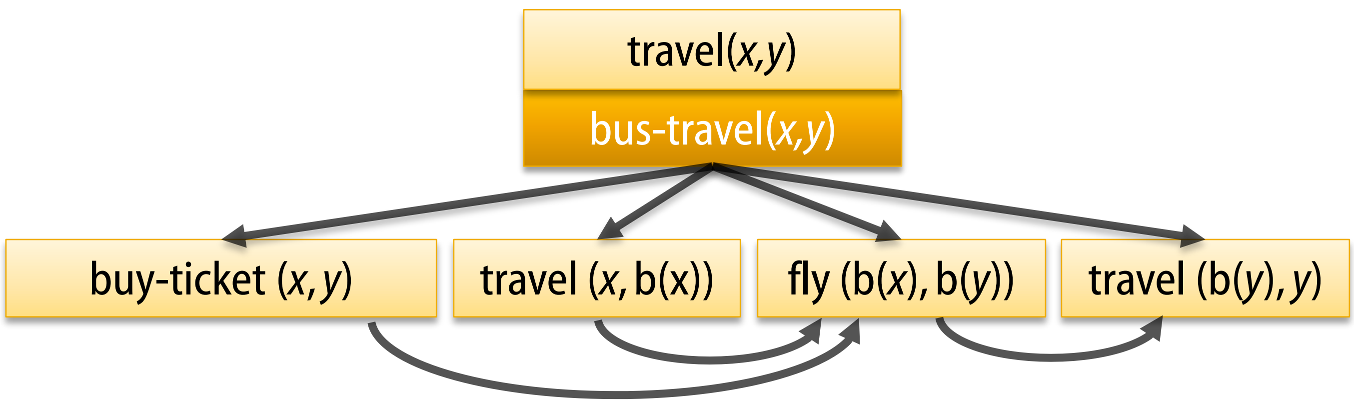


The Delta Function (1)

- What does $\delta(w, u, m, \sigma)$ mean?
  - We picked a task that **could be first**

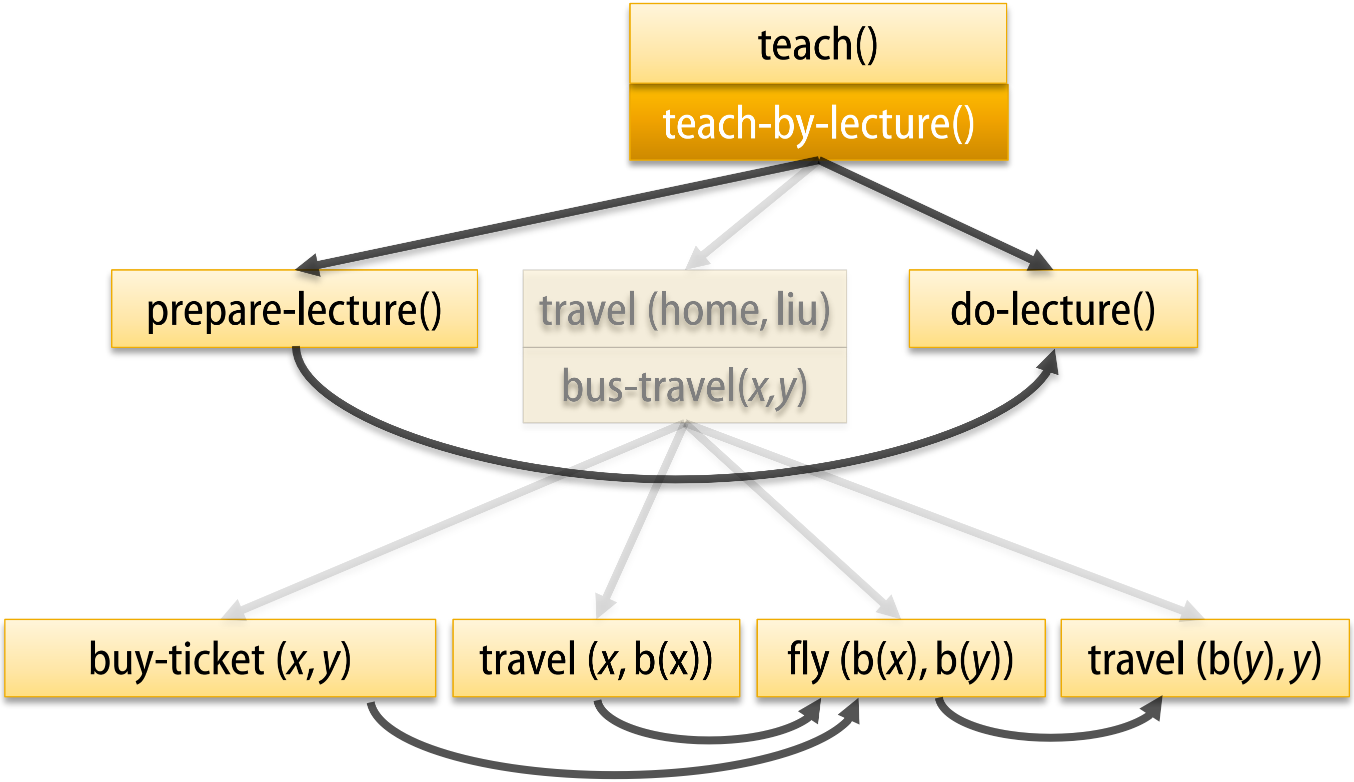


  - We picked a partial-order **decomposition** of that task



  - First, we **replace** the selected task with its expansion

The Delta Function (2)



The Delta Function (3)

- Second, the method itself can have **preconditions**
  - We have tested the preconditions, and they hold
  - We must make sure they **still** hold when the first **subtask** is executed

- Must do u's first subtask before the first subtask of every $t_i \neq$ u
  - The first subtask of bus-travel
    before the first subtask of prepare-lecture

- But which one is first? It's partially ordered, so we don't know!
  - So δ creates one alternative for each possible "first" subtask of u
    - In our case, buy-ticket or travel (x, b(x)) can be first

  - Then we nondeterministically choose between these alternatives

Partially Ordered Methods

- Note that **only methods** are partially ordered
  - The problem specification does not have to define the exact execution order in advance
- The **final plan** is totally ordered!
  - The planner chooses an order

Expressivity

Comparison to Classical Planning

Any classical problem

Polynomial-time transformation

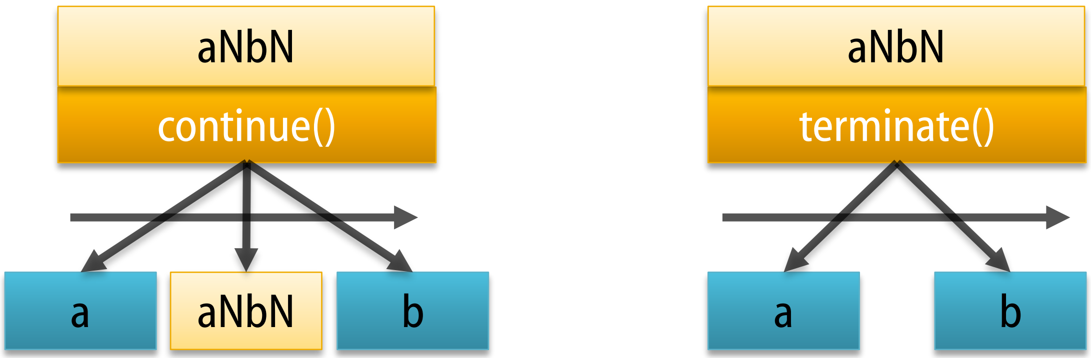
Corresponding STN problem

For some STN problems,
there **exists no classical problem** with the same set of solutions!

Even Simple Task Networks
are **strictly more expressive** than classical planning

Comparison to Classical Planning

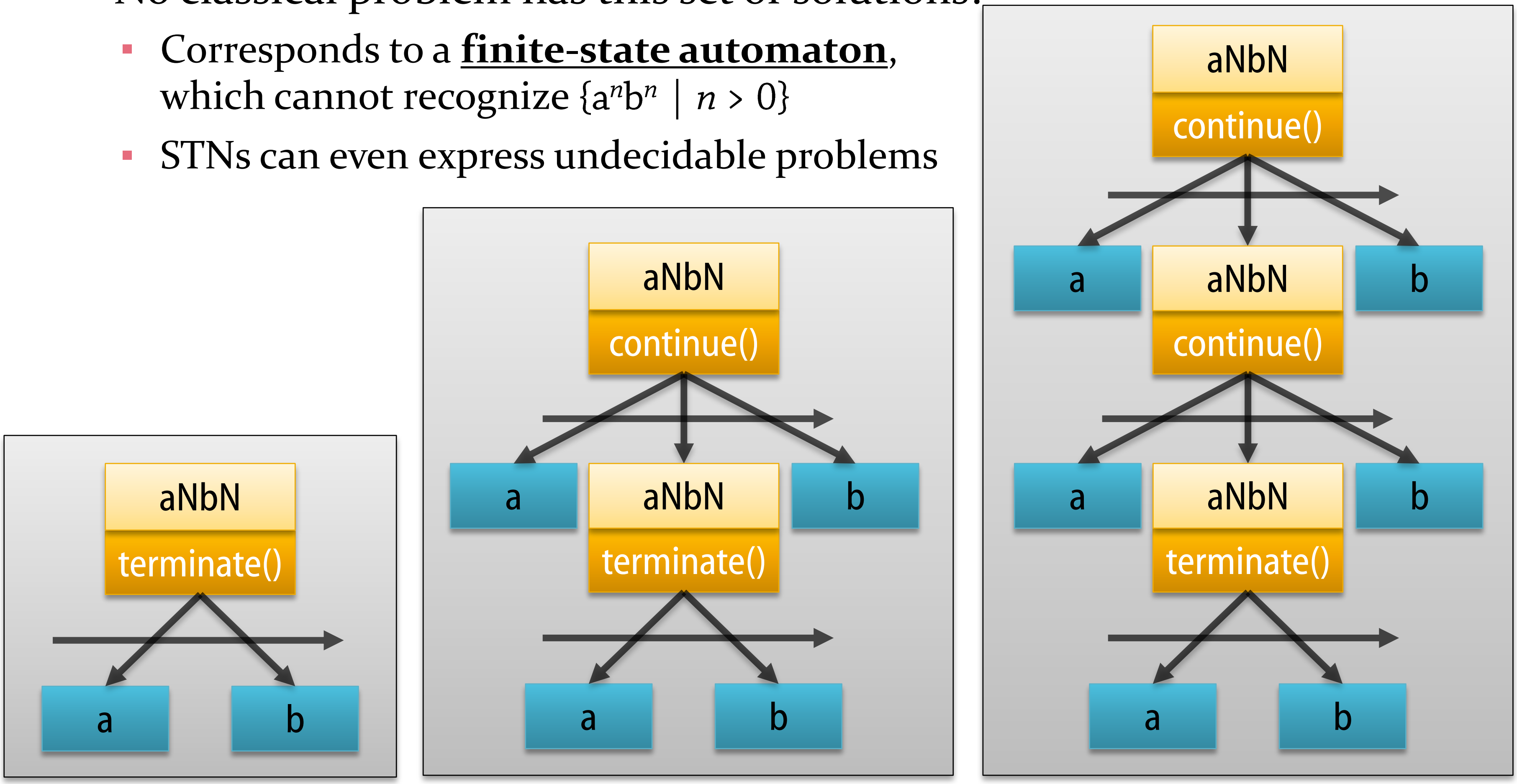
- Artificial example:
  - Two primitive tasks, a and b
  - Two STN methods:



  - s0 = {}
  - Initial task:

Comparison to Classical Planning

- Possible solutions:
  - $\{a^n b^n \mid n > 0\}$
  - No classical problem has this set of solutions!
    - Corresponds to a **<u>finite-state automaton</u>**, which cannot recognize $\{a^n b^n \mid n > 0\}$
    - STNs can even express undecidable problems



Conclusion

Contrast: HTN

- Control Rules or Hierarchical Task Networks?
  - Both can be very efficient and expressive

  - If you have "**recipes**" for everything, HTN can be more convenient
    - **Can** be modeled with control rules, but not intended for this purpose
    - You have to forbid everything that is "outside" the recipe

  - If you have knowledge about "**some things that shouldn't be done**":
    - With control rules, the default is to "try everything"
    - Can more easily express localized knowledge
      about what should and shouldn't be done
    - Doesn't require knowledge of all the ways in which the goal can be reached