Automated Planning

Domain-Configurable Planning: Hierarchical Task Networks

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

jonas.kvarnstrom@liu.se – 2018

HTNs: Ideas

jonkv@ida

- Classical Planning vs. Hierarchical Task Networks:

Classical Planning		Hierarchical Task Networks
Objective is to **achieve a goal**: **at**(TimesSquare) { **on**(A,B), **on**(C,D) } …	→	Objective is to **perform a task**: **travel-to**(TimesSquare) **place-blocks-correctly** …
Find any sequence of actions that achieves the goal	→	Use "templates" to incrementally **refine** the task until you reach primitive actions **travel-to**(TimesSquare) ➔ **taxi-to**(airport); **fly-to**(JFK); …

Provides **guidance** but still requires **planning**

Total-Order
Simple Task Networks

A simple form of Hierarchical Task Network,
as defined in the book

Terminology 1: Primitive Task

- A **primitive task** is an **action**
  - Anything that can be directly executed

stack(A,B)

load(crane1, loc3, cont5, …)

point(camera4, obj4)

Dark green
(in this presentation):
"Done", no need to think further

  - As in classical planning, **what is primitive** depends on:
    - The ***execution system***
    - ***How detailed*** you want your plans to be
  - Example:
    - For you, **fly(here,there)** may be a primitive task
    - For the pilot, it may be decomposed into many smaller steps
  - Tasks can be *ground* or *non-ground*: **stack(A,?*x*)**
    - No separate terminology, as in *operator/action*

Terminology 2: Non-Primitive Task

- A **non-primitive task**:
  - Cannot be directly executed
  - Must be **decomposed** into 0 or more **subtasks**

put-all-blocks-in-place()

make-tower(A,B,C,D,E)

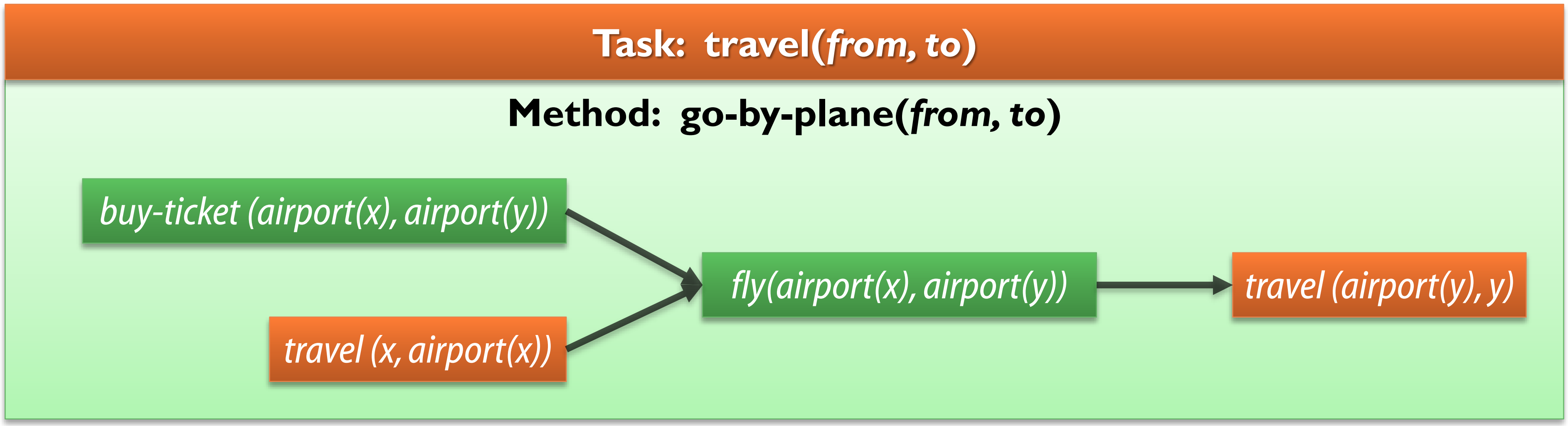
move-stack-of-blocks(x, y)

Orange:
There's a "problem"
that we need to solve

Should be *decomposed to*
pickup, putdown, stack, unstack
tasks / actions!

Terminology 3: Method

- A **method** specifies *one way* to decompose a non-primitive task



- The decomposition is a **graph** $\langle N, E \rangle$
  - Nodes in N correspond to **subtasks to perform**
    - Can be primitive or not!
  - Edges in E correspond to **ordering relations**

Totally Ordered STNs

In **Totally Ordered Simple Task Networks (STN),**
each method must specify a **sequence** of subtasks

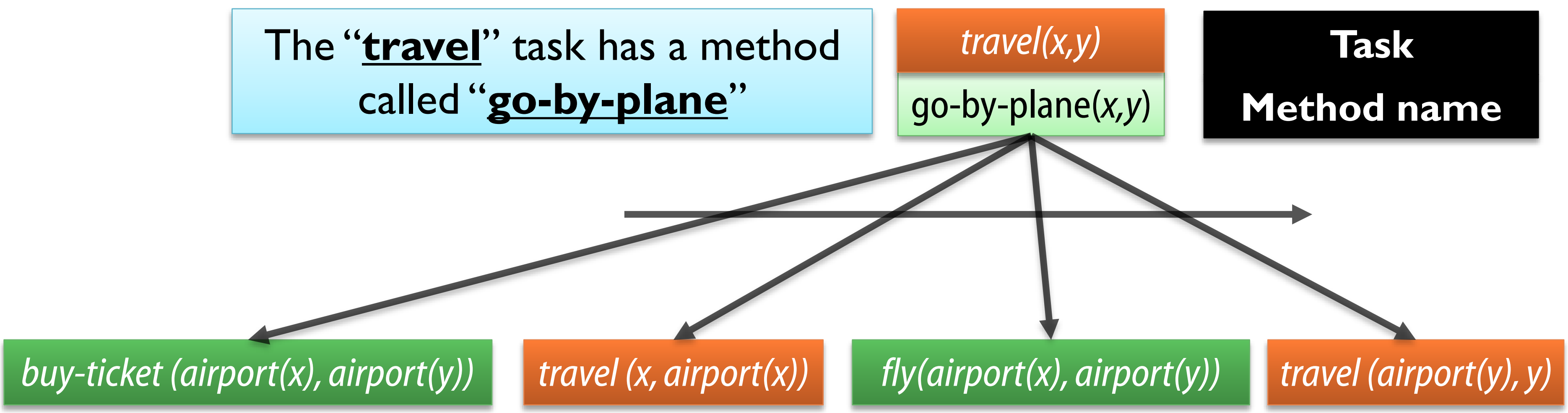
- Can still be modeled as a graph $\langle N, E \rangle$

buy-ticket (airport(x), airport(y)) → *travel (x, airport(x))* → *fly(airport(x), airport(y))* → *travel (airport(y), y)*

- Alternatively: A sequence $< t_1, \ldots, t_k >$
  - <**buy-ticket**(airport(x), airport(y)),
    travel(x, airport(x)),
    fly(airport(x), airport(y)),
    travel(airport(y), y) >

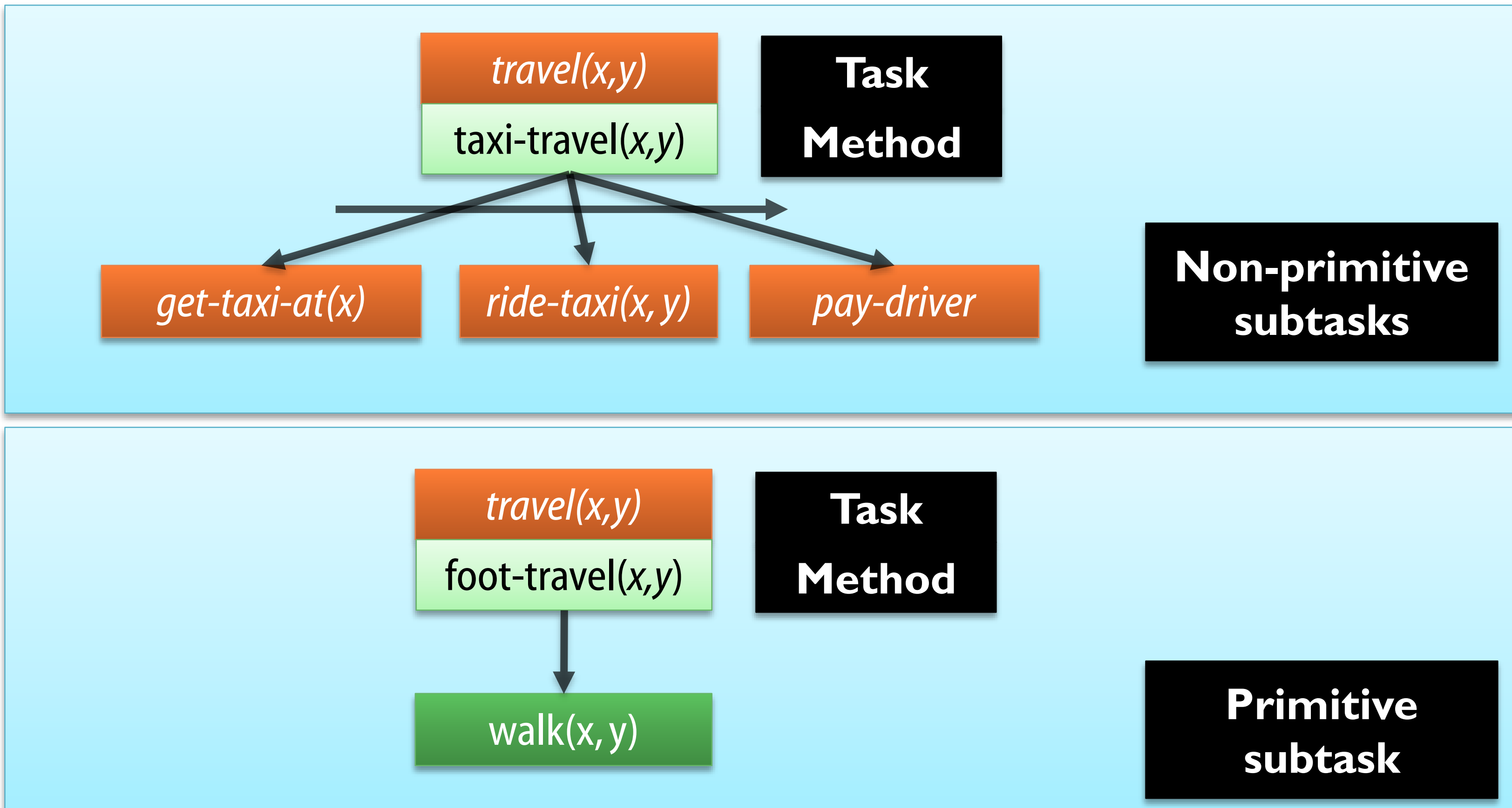
Totally Ordered STNs (2)

We can **illustrate** the **entire decomposition** in this way
(horizontal arrow → sequence)



Multiple Methods

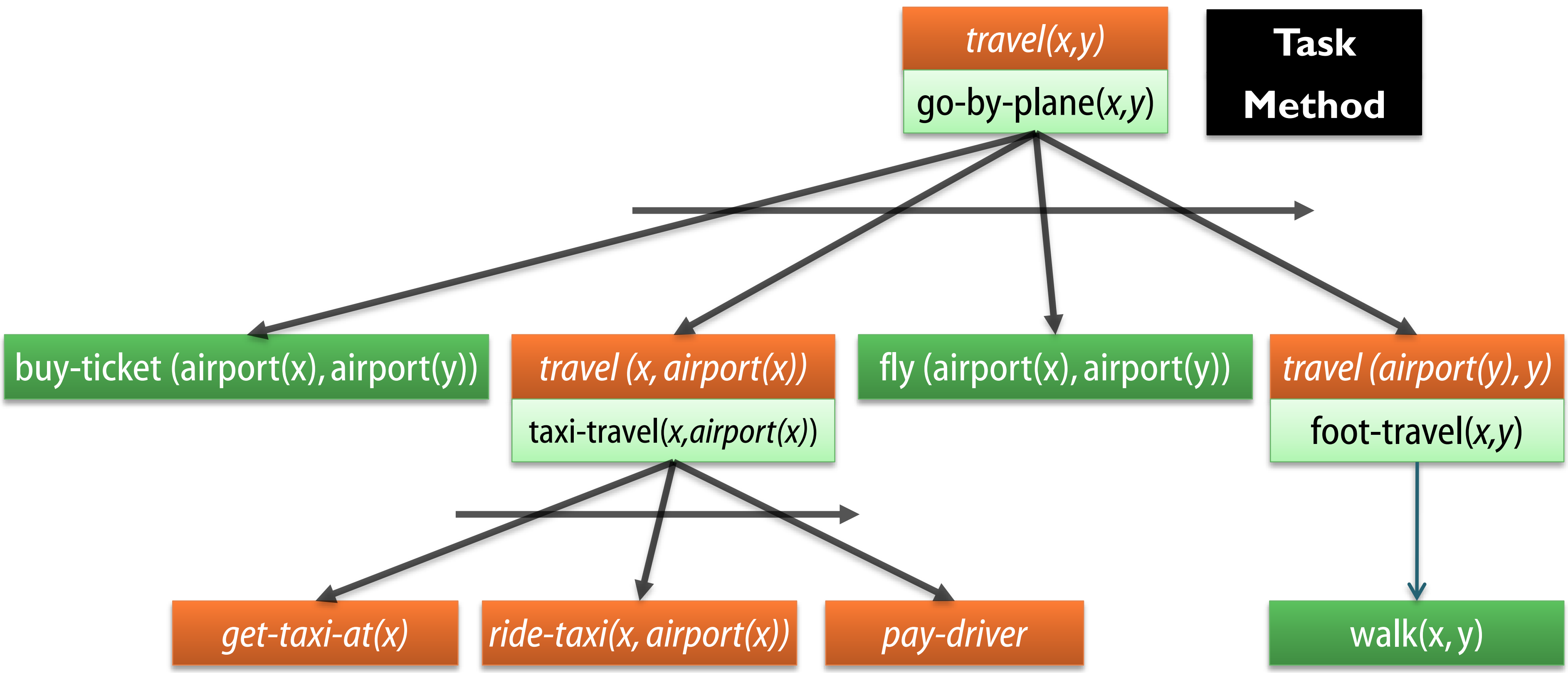
- A non-primitive task can have **<u>many methods</u>**
  - So: You still need to **<u>search</u>**, to determine which method to use



  - …and to determine *parameters* (shown later)

Composition

- An HTN plan:
  - **Hierarchical**
  - Consist of **tasks**
  - Based on graphs ≈ **networks**



Domains, Problems, Solutions

jonkv@ida

- An STN **planning domain** specifies:
  - A set of **tasks**
  - A set of **operators** used for primitive tasks
  - A set of **methods**

General HTNs: Can have additional constraints to be enforced

- An STN **problem instance** specifies:
  - An STN planning domain
  - An **initial state**
  - An **initial task network**, which should be ground (no variables)
    - Total Order STN example:
      <travel(home,work); do-work(); travel(work,home)>

Domains, Problems, Solutions (2)

- Suppose you:
  - Start with the **initial task network**
  - Recursively apply **methods** to non-primitive tasks, expanding them
  - Continue until **all non-primitive tasks are expanded**

- Totally ordered → yields an action **sequence**
  - If this is executable: A **solution**
  - (No goals to check – implicit in the method structure!)

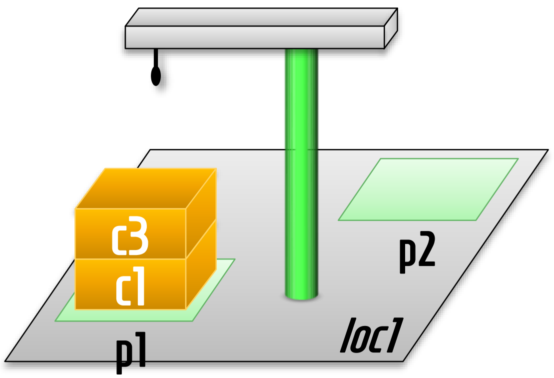
- The planner uses **only** the methods specified for a given task
  - Will **not** try arbitrary actions…
  - For this to be useful, you must **have** useful "recipes" for all tasks

DWR Example: Moving the Topmost Container

A simple "template expansion"

DWR

- Let's switch to Dock Worker Robots…



- Example Tasks:
  - Primitive – all DWR actions
  - Move the **topmost** container between piles
  - Move an **entire stack** from one pile to another
  - Move a stack, but keep it in the **same order**
  - Move **several stacks** in the same order

Methods

jonkv@ida

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

The *task* has parameters **given from above**

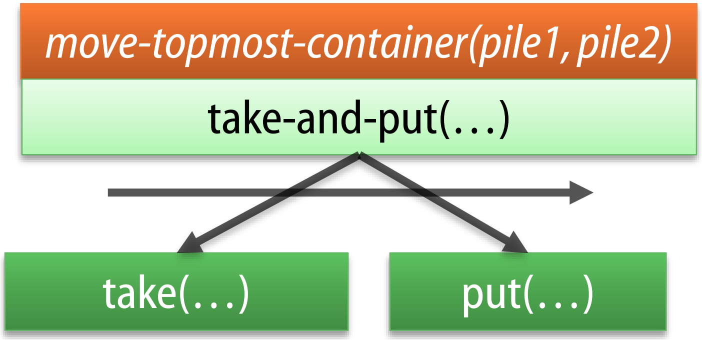
A *method* can have additional parameters, whose values are **chosen by the planner** – just as in classical planning!

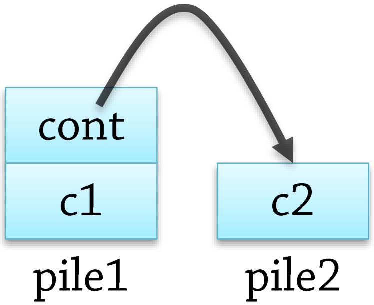
The *precond* adds constraints: crane must be *some* crane in the same loc as the piles, cont must be *the* topmost container of pile1, …

Interpretation:
If you are asked to **move-topmost-container**(pile1, pile2),
check all possible values for **cont, crane, loc, c1, c2** where the preconds are satisfied

Methods (2)

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





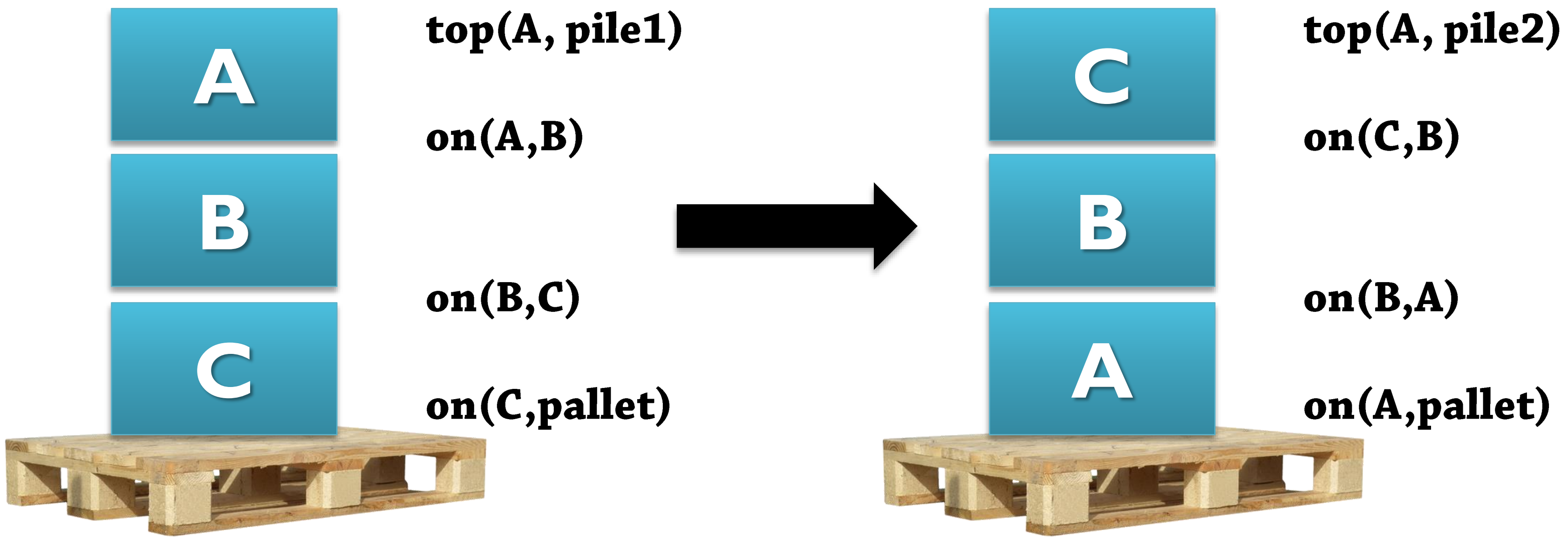
DWR Example: Moving a Stack of Containers

Iteration with no predetermined bound

Moving a Stack of Containers

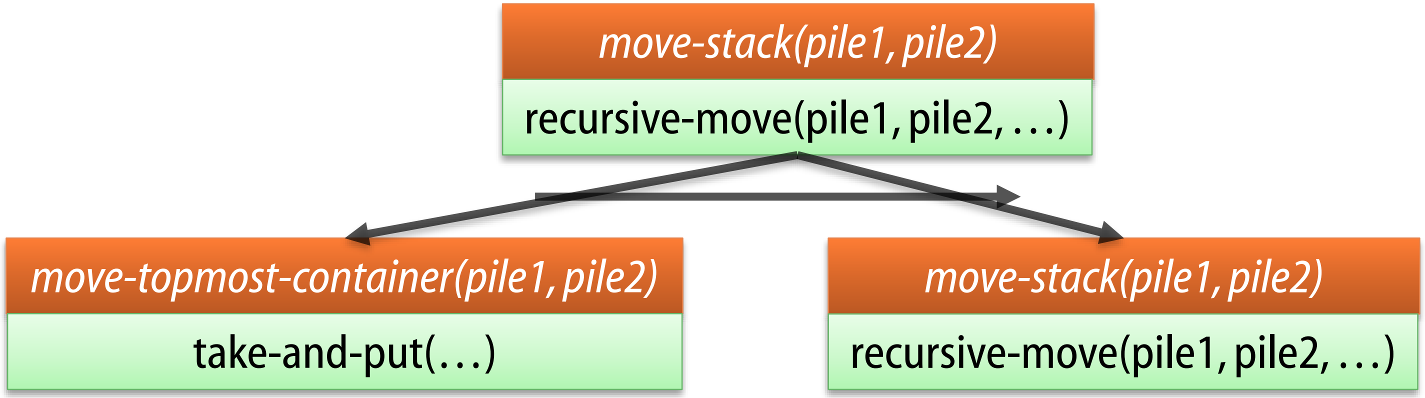
jonkv@ida

- How can we implement the task **move-stack**(pile1, pile2)?
  - Should move **all** containers in a stack
  - There is no **limit** on how many there might be…



Recursion (1)

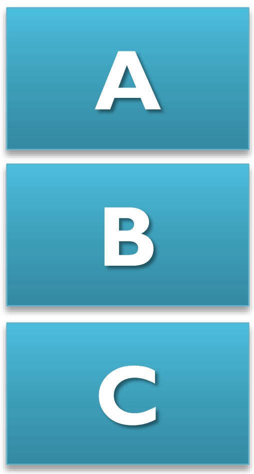
- We need a **loop** with a **termination condition**
  - HTN planning allows **recursion**
    - Move the **topmost** container (we know how to do that!)
    - Then move the **rest**
  - First attempt:
    - **task:** move-stack(pile1, pile2)
    - **method**: recursive-move(pile1, pile2)
    - **precond**: true
    - **subtasks**: <move-topmost-container(pile1, pile2), move-stack(pile1, pile2)>



Recursion (2)

- But consider the BW and DWR "pile models"…

BW	DWR

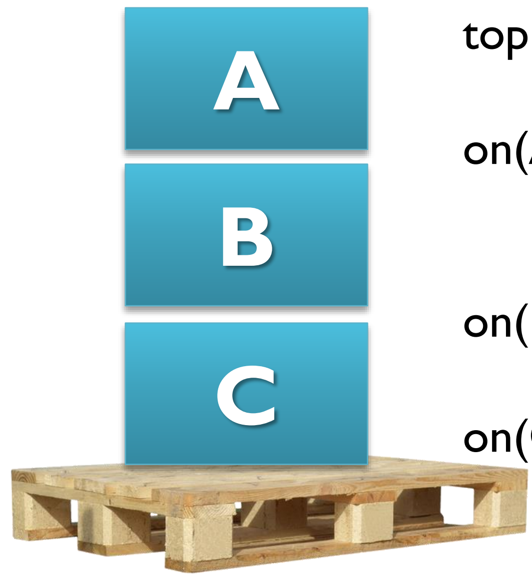


clear(A)

on(A,B)

on(B,C)

ontable(C)



top(A, pile1)

on(A,B)

on(B,C)

on(C,pallet)

The bottom block
is not "on" anything

The bottom block
is "on" the pallet, a "special container"

What if the pallet is "topmost"?
We don't want to move it!

Recursion (3)

- To fix this:

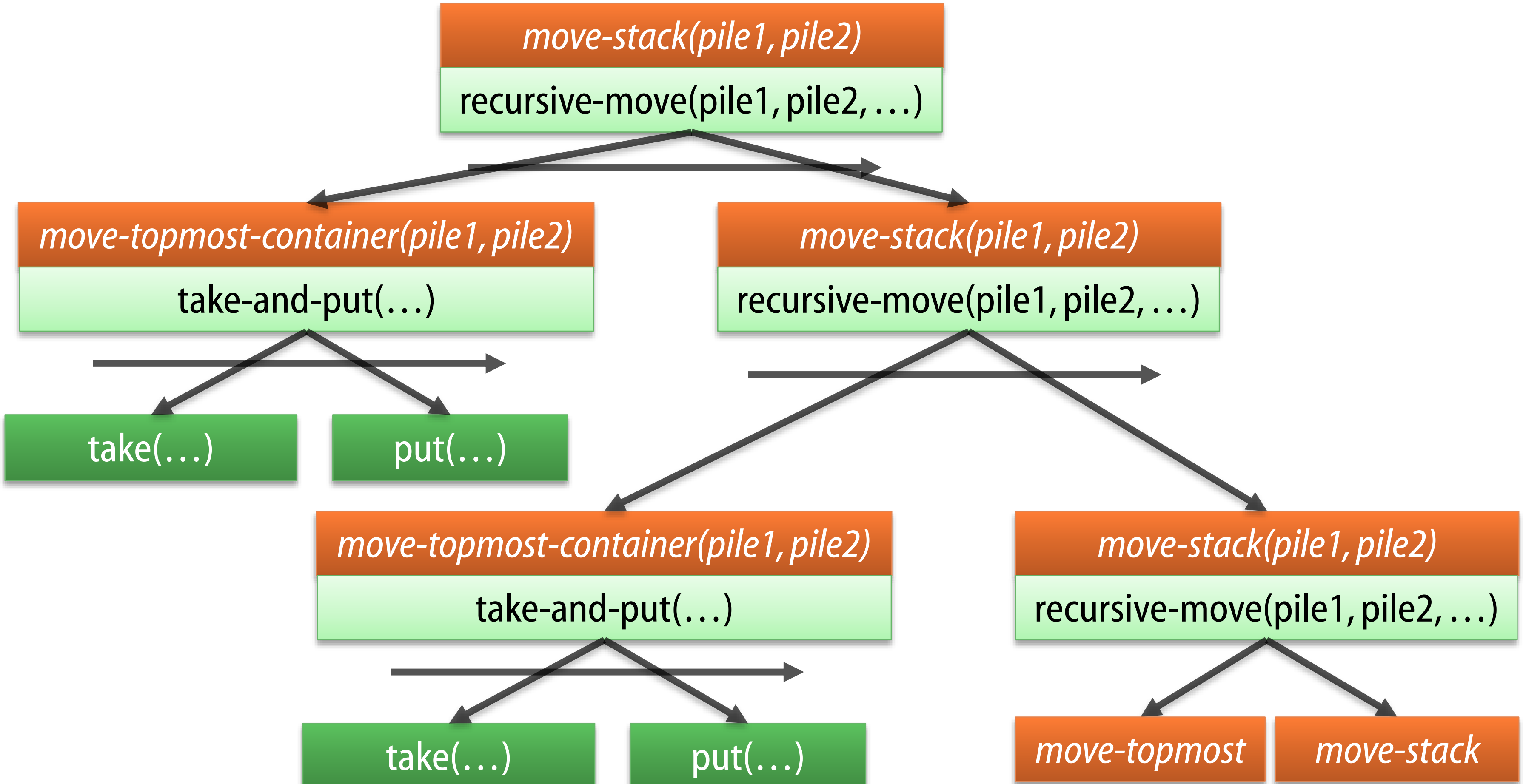
Add two method params – "non-natural", as in "ordinary" planning; does not give the planner a real choice

  - **Task**: move-stack(pile1, pile2)
    - **method**: recursive-move(pile1, pile2, ***cont***, ***x***)
    - **precond**: **top(*cont*, pile1), on(*cont*, *x*)**
    - **subtasks**: <move-topmost-container(pile1, pile2), move-stack(pile1, pile2)>

cont **is on top of something (x), so** ***cont*** **can't be the pallet**

Recursion (4)

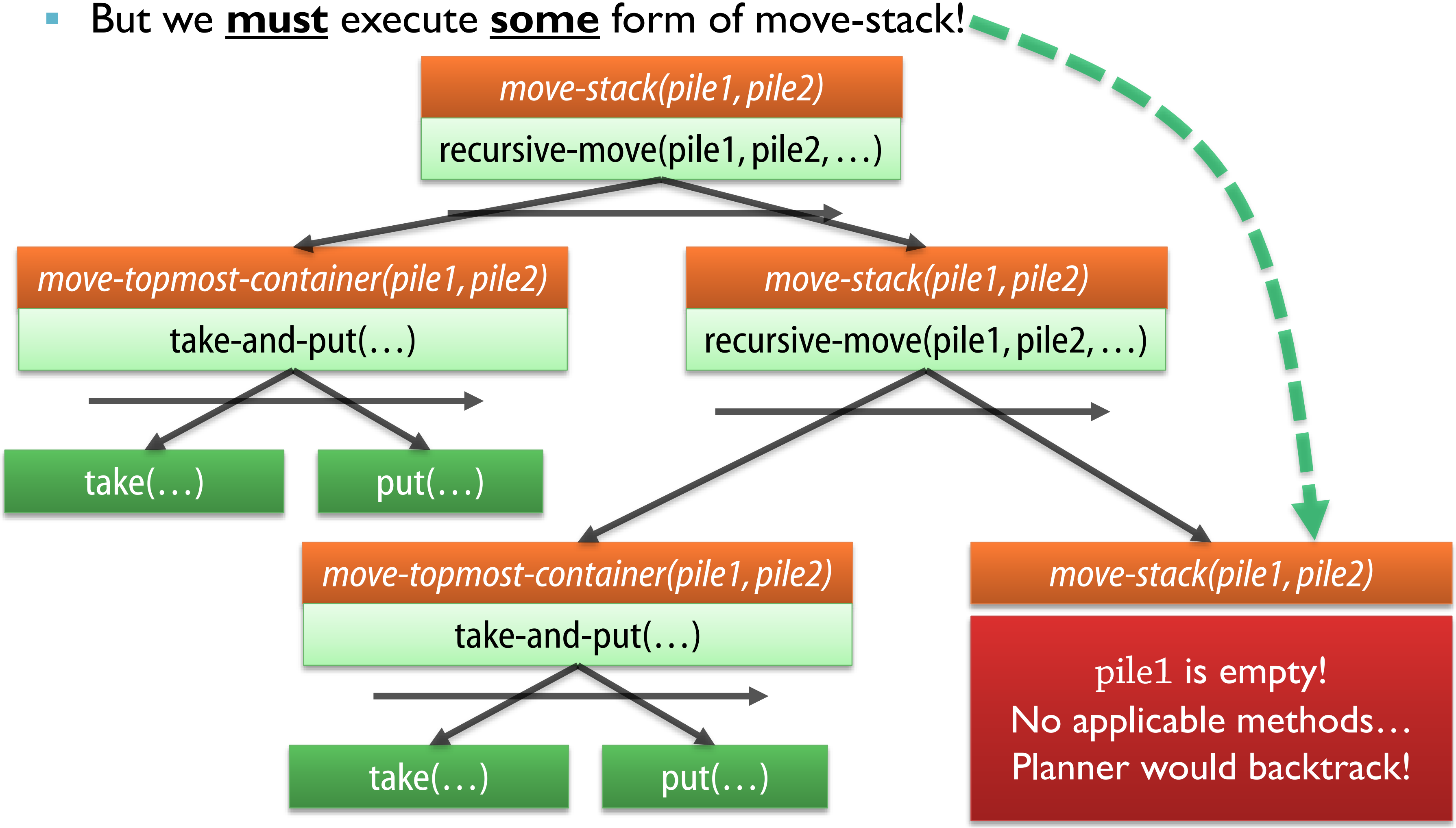
- The planner can now create a structure like this:



But when will the recursion end?

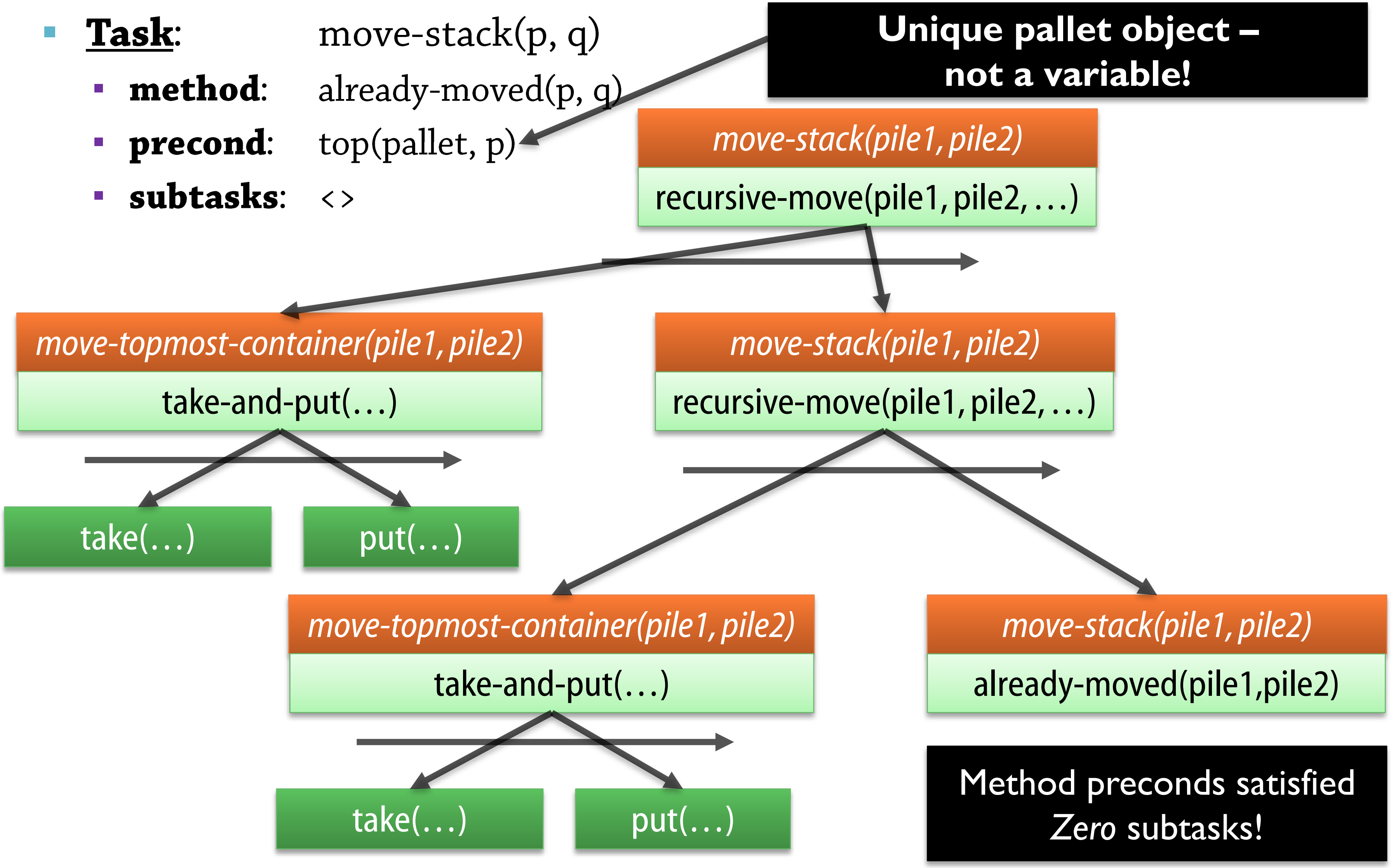
Recursion (5)

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



Recursion (6)

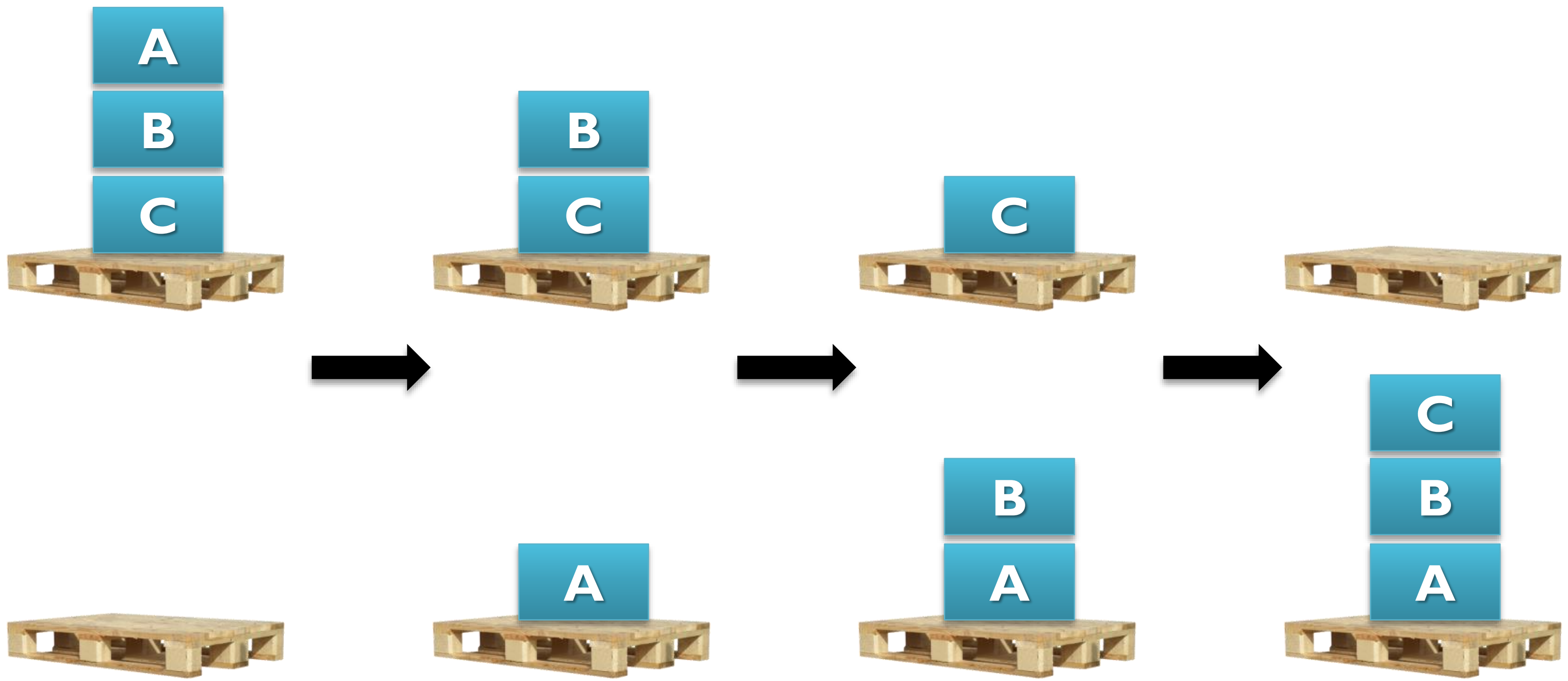
- We need a method that **terminates** the recursion
  - **Task**: move-stack(p, q)
    - **method**: already-moved(p, q)
    - **precond**: top(pallet, p)
    - **subtasks**: <>



DWR Example:
Moving a stack, in the same order

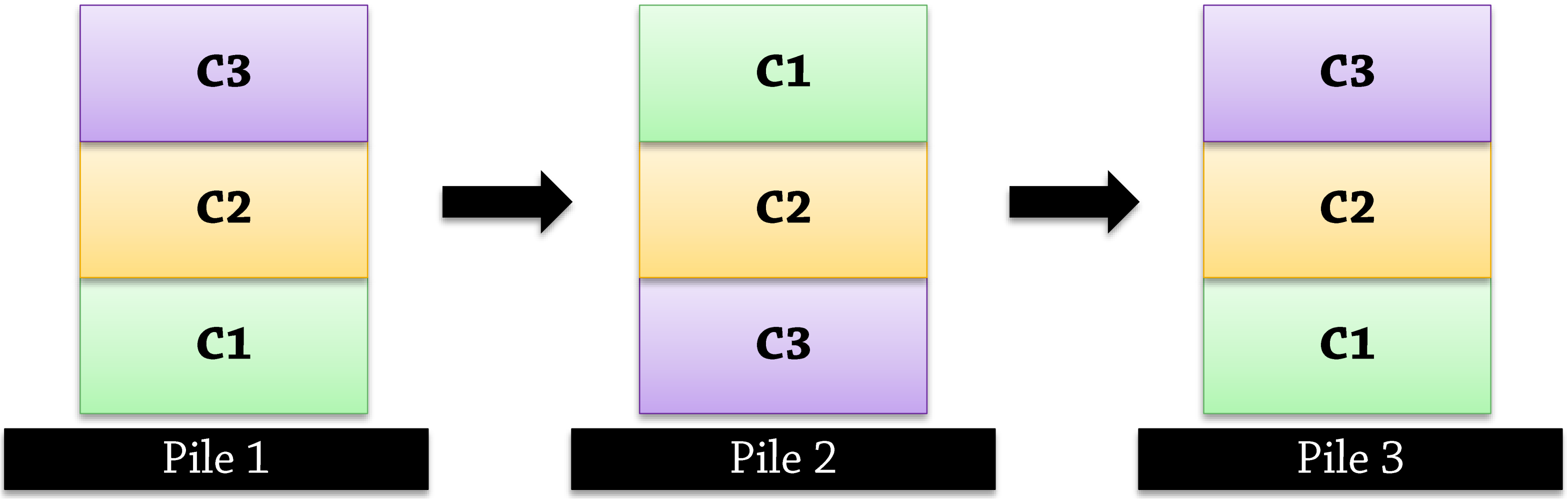
Ordering (1)

- Using move-stack inverts a stack:



Ordering (2)

- To avoid this: Use an intermediate pile



Ordering (3)

- Example:
  - **Task:** move-stack-same-order(pile1, pile2)
    - **method**: move-each-twice(**pile1, pileX, pile2, loc**)
    - **precond**: top(pallet, pileX),
      pile1 != pileX, pile2 != pileX, pile1 != pile2,
      attached(…), // All in the same location
      …
    - **subtasks**: ; move twice:
      <move-stack(pile1, **pileX**), move-stack(**pileX**, pile2)>

Planner *chooses* pileX, *finds* location

Why does **pileX** have to be empty initially?

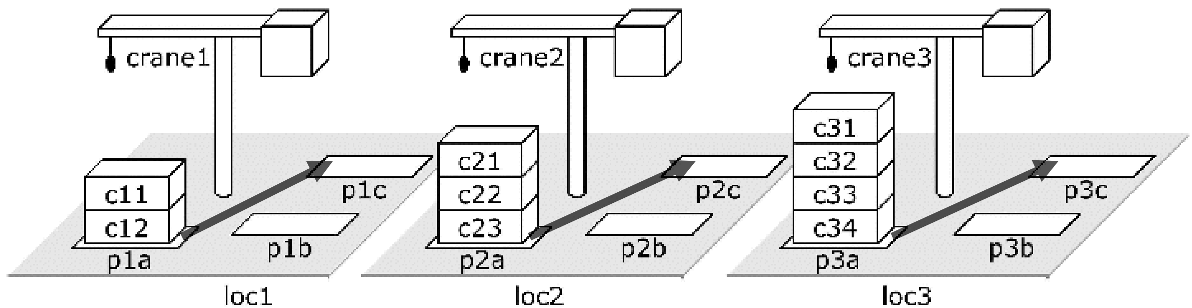
Because the second *move-stack* moves *all* containers from the intermediate pile…

DWR Example:
Moving Three Stacks

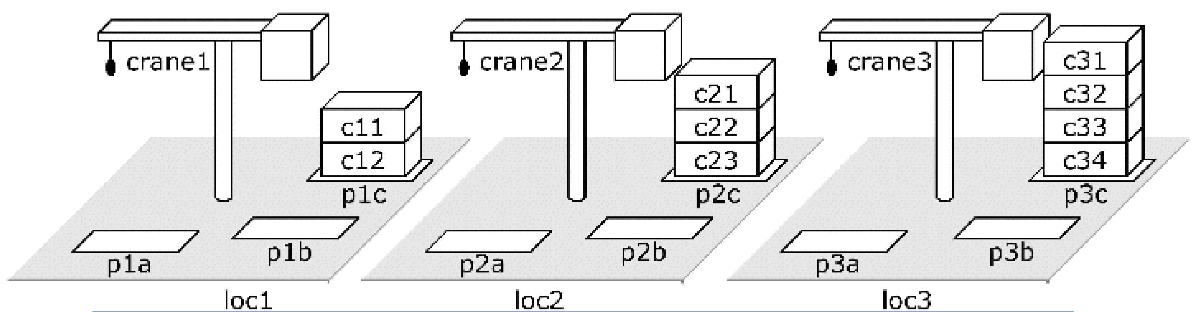
Letting the planner choose *parameters*

Overall Objective

- Our overall **objective** is:
  - Moving three entire stacks of containers, preserving order



Initial state, with 3 locations, 3 piles to move

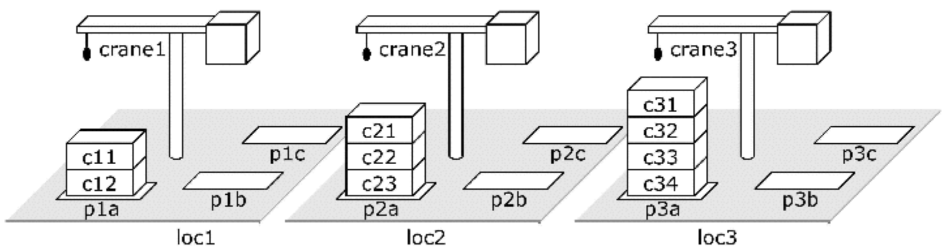


Corresponding objective, all piles moved

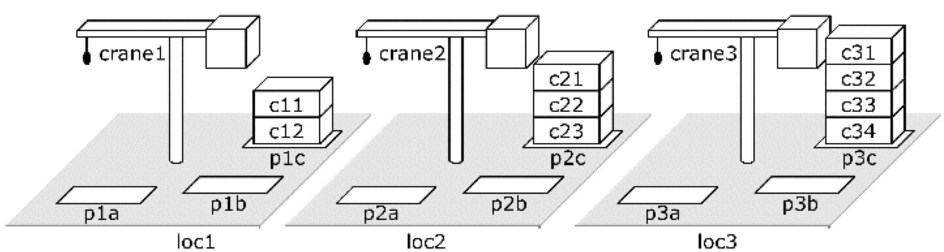
Overall Objective: Defining a Task

- Define a **task** for this objective
  - **Task:** move-three-stacks()
    - **method**: move-each-twice()
    - **precond**: ; no preconditions apart from the subtasks'
    - **subtasks**: ; move each stack twice:
      <move-stack-same-order(p1a,p1c),
      move-stack-same-order(p2a,p2c),
      move-stack-same-order(p3a,p3c) >

- Use this task
  as the *initial task network*



(a) initial state



(b) goal

DWR Example: Moving *n* stacks

Letting the planner choose *parameters*

Goal Predicates in HTNs

- Here the **entire** objective was encoded in the initial network
  - **move-three-stacks**
    - ➔ <move-stack-same-order(p1a,p1c),
      move-stack-same-order(p2a,p2c),
      move-stack-same-order(p3a,p3c) >
- To avoid this:
  - New predicate ***should-move-same-order****(pile, pile)* encoding the goal
  - **Task:** move-as-necessary()
    - **method**: **move-and-repeat**(pile1, pile2)
    - **precond**: should-move-same-order(pile1, pile2)
    - **subtasks**: <move-stack-same-order(pile1, pile2), ;; makes should-move… false!
      move-as-necessary>
  - **Task:** move-as-necessary()
    - **method**: **all-done**
    - **precond**: not exists pile1, pile2 [should-move-same-order(pile1, pile2)]
    - **subtasks**: <>

Uninformed Planning in HTNs

- Can even do **uninformed unguided planning**
  - Doing *something, anything*:
    - Task **do-something** → operator **pickup(x)**
    - Task **do-something** → operator **putdown(x)**
    - Task **do-something** → operator **stack(x,y)**
    - Task **do-something** → operator **unstack(x,y)**

Planner chooses all parameters

  - Repeating:
    - Task **achieve-goals** → <do-something, achieve-goals>
  - Ending:
    - Task **achieve-goals** → <>, with precond: entire goal is satisfied

Or combine **aspects** of this model
with **other aspects** of "standard" HTN models!

Useful Modeling Strategies:

Delivery Example – Delivering a package

Modeling "conditional" actions

Delivery 1: First Variation

jonkv@ida

- **Delivery**:
  - A single truck
  - Pick up a package, drive to its destination, unload

  - **Task:** **deliver**(*package*, *dest*)
    - **method**: **move-by-truck**(*package*, *packageloc*, *dest*)
    - **precond**: **at**(*package*, *packageloc*)
    - **subtasks**: <**driveto**(*packageloc*), **load**(*package*), **driveto**(*dest*), **unload**(*package*)>

What if the truck is already *at* the package location?
First driveto is unnecessary!

Delivery 2: Second Variation

- **Alternative**: Two alternative methods for *deliver*

  - **Task:** deliver(*package*, *dest*)
    - **method**: move-by-truck-1(*package*, *packageloc*, *truckloc*, *dest*)
    - **precond**: at(truck, *truckloc*), at(*package*, *packageloc*),
      packageloc = truckloc
    - **subtasks**: <load(*package*), driveto(*dest*), unload(*package*)>

  - **Task**: deliver(*package*, *dest*)
    - method: move-by-truck-2(*package*, *packageloc*, *truckloc*, *dest*)
    - precond: at(*truck*, *truckloc*), at(*package*, *packageloc*),
      packageloc != truckloc
    - subtasks: <**driveto(*packageloc*)**,
      load(*package*), driveto(*dest*), unload(*package*)>

Do we really have to repeat the entire task?
Many "conditional" subtasks → combinatorial explosion

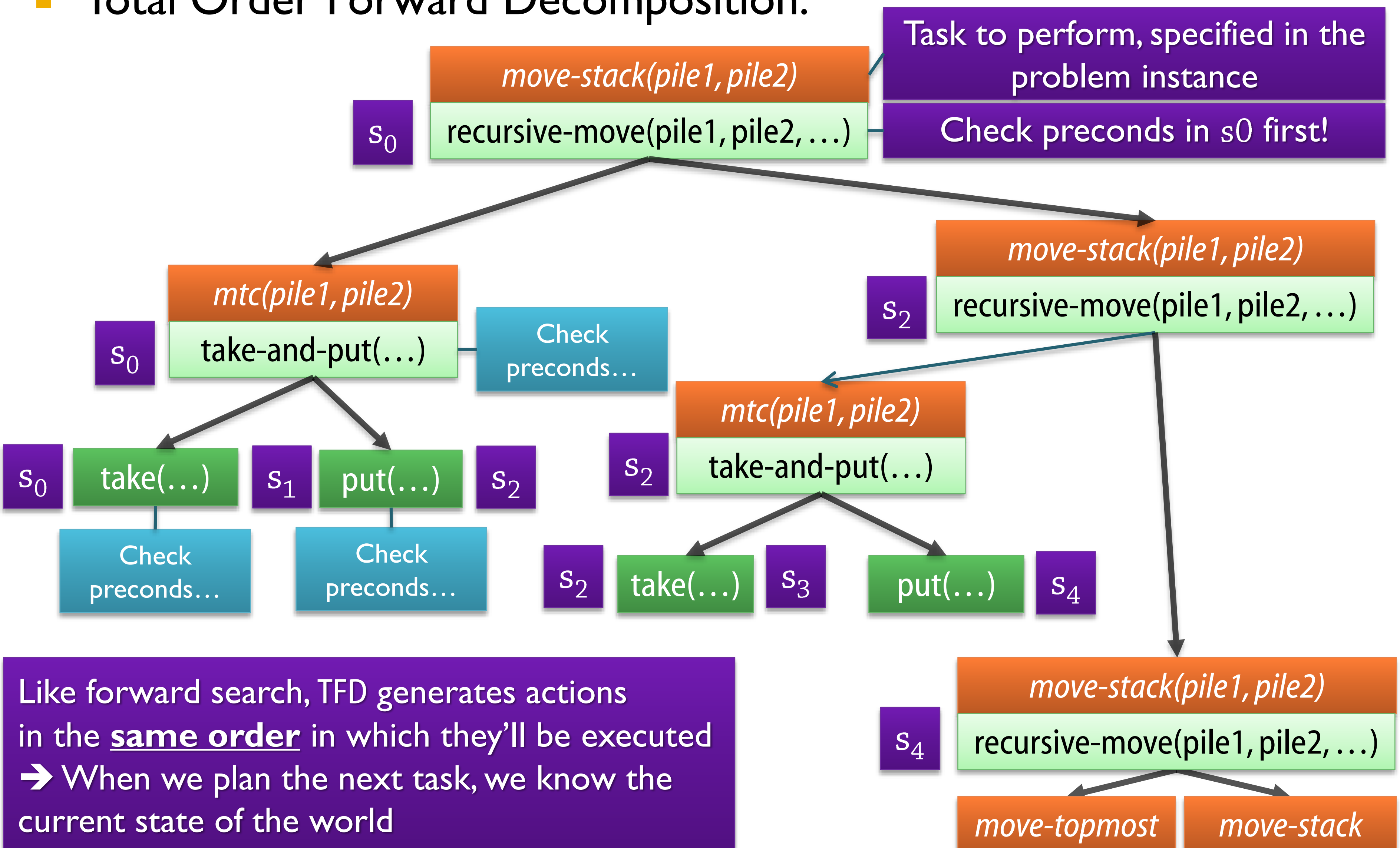
Delivery 3: Third variation

- Make the choice in the subtask instead!
  - **Task**: deliver(*package*, *dest*)
    - **method**: move-by-truck-3(*package*, *packageloc*, *truckloc*, *dest*)
    - **precond**: at(truck, *truckloc*), at(*package*, *packageloc*)
    - **subtasks**: <**be-at**(*packageloc*), load(*package*), **be-at**(*dest*), unload(*package*)>
  - **Task**: **be-at**(*loc*)
    - **method**: drive(*loc*)
    - **precond**: !at(truck, *loc*)
    - **subtasks**: <driveto(*loc*)>
  - **Task**: **be-at**(*loc*)
    - **method**: already-there
    - **precond**: at(truck, *loc*)
    - **subtasks**: < >

A Planning Algorithm:
Total Order Forward Decomposition

Total Order Forward Decomposition

- Total Order Forward Decomposition:

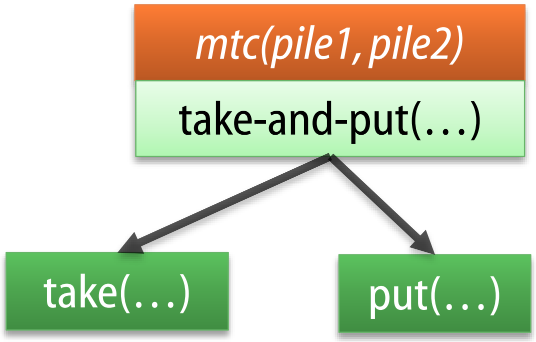


Like forward search, TFD generates actions in the **same order** in which they'll be executed
➔ When we plan the next task, we know the current state of the world

Definitions

- Primitive Tasks vs. Operators:
  - We've said…
    - A primitive task *is* an action

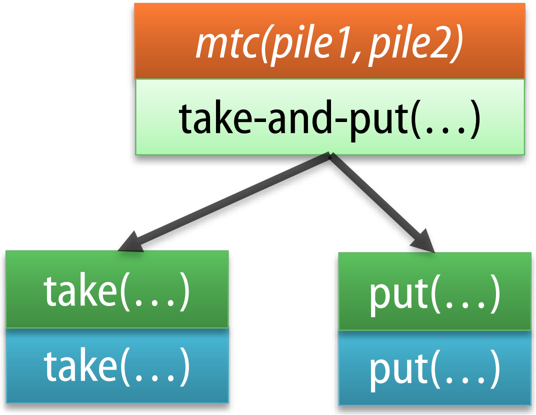
Primitive task = action



  - The book says…
    - A primitive task *is decomposed to* a single action

Primitive task

Action



Not an essential difference,
as long as you are consistent!

Solving Total-Order STN Problems (1)

- TFD takes an STN problem instance:
  - s – the current state
  - <t1,…,tk> – a list of tasks to be achieved **in the specified order**
  - O – the available operators (with params, preconds, effects)
  - M – the available methods (with params, preconds, subtasks)

- **Returns**:
  - A sequential plan
    - Loses the hierarchical structure of the final plan
    - Simplifies the presentation – but the structure *could* also be kept!

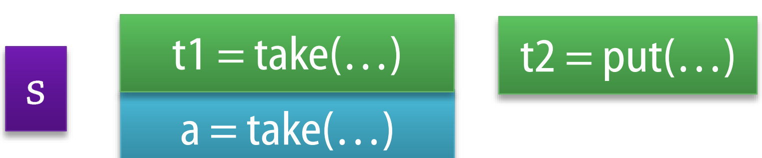
  - **TFD(s, <t1,…,tk>, O, M):**
    - // If we have no tasks left to do…
      if (k = 0) **then** return the empty plan

Solving Total-Order STN Problems (2)

For simplicity: The case where all **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!
    // May be many to choose from (e.g. method has more params than task).
    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



Waiting in line
to be decomposed
in the next step

Solving Total-Order STN Problems (3)

For simplicity: The case where all tasks are ground

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

    // 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 | a ∈ *actions* and
    a is relevant for t1 and // Achieves the task
    a is applicable in s }
    if (*candidates* = ∅) return failure

  - **nondeterministically choose** any a ∈ *candidates* // Or use backtracking
  - *newstate* ← $\gamma(s, a)$ // Apply the action, find the new state

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

Solving Total-Order STN Problems (4)

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

    // 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 }
    if (*candidates* = ∅) return failure

The case where tasks are **non-ground**: move(container1,*X*)

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

(italics = variables)

s

t1 = take(*crane*, loc1, cont2, *cont*, pile8)

t2=put(*crane*, …)

candidates

take(**crane1**, loc1, cont2, **cont5**, pile8) — σ = { *crane*↦crane1, *cont*↦cont5 }

take(**crane2**, loc1, cont2, **cont5**, pile8) — σ = { *crane*↦crane2, *cont*↦cont5 }

Solving Total-Order STN Problems (5)

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

    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 backtracking
  - *newstate* ← $\gamma(s, a)$ // 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

(italics = variables)

s

t1 = take(*crane*, loc1, cont2, *cont*, pile8)

chosen: a = take(<u>crane1</u>, loc1, cont2, <u>cont5</u>, pile8) σ = { *crane*↦crane1, *cont*↦cont5 }

take(<u>crane2</u>, loc1, cont2, <u>cont5</u>, pile8) { *crane*↦crane2, *cont*↦cont5 }

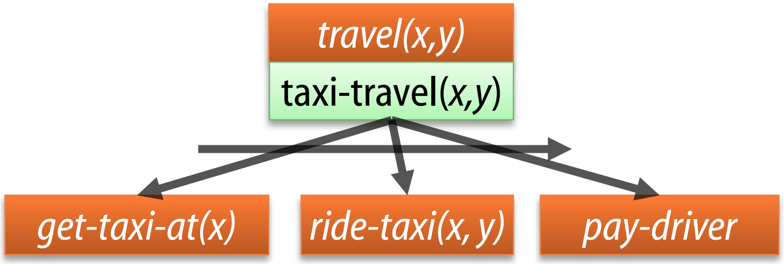
σ(t2) = put(<u>crane1</u>, …)

t2=put(*crane*, …)

Solving Total-Order STN Problems (6)

- 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) ∈ *candidates* // Or use backtracking

As before, but methods instead of actions

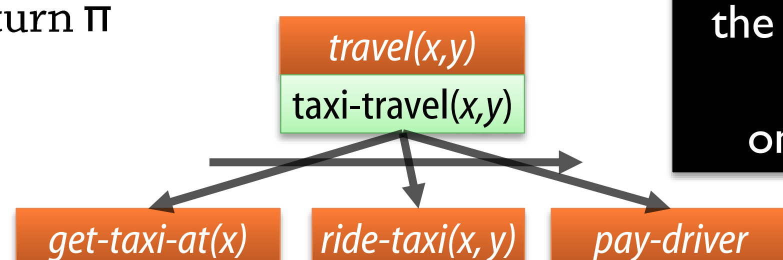


Solving Total-Order STN Problems (7)

- 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

Replace the task by its subtasks

// No actions are applied here, so no new state!
remaining ← **subtasks(m)** . σ(<t2,…,tk>) // Prepend new list!
π ← *TFD(s, remaining*, O, M)
if (π = failure) return failure
else return π



In TFD
the "origin" of a task is discarded:
No longer needed,
only the subtasks are relevant

Limitations of
Total-Order HTN Planning

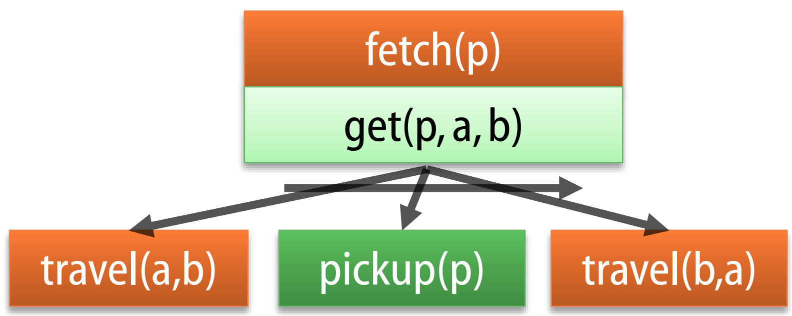
Limitation of Ordered-Task Planning

- 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)

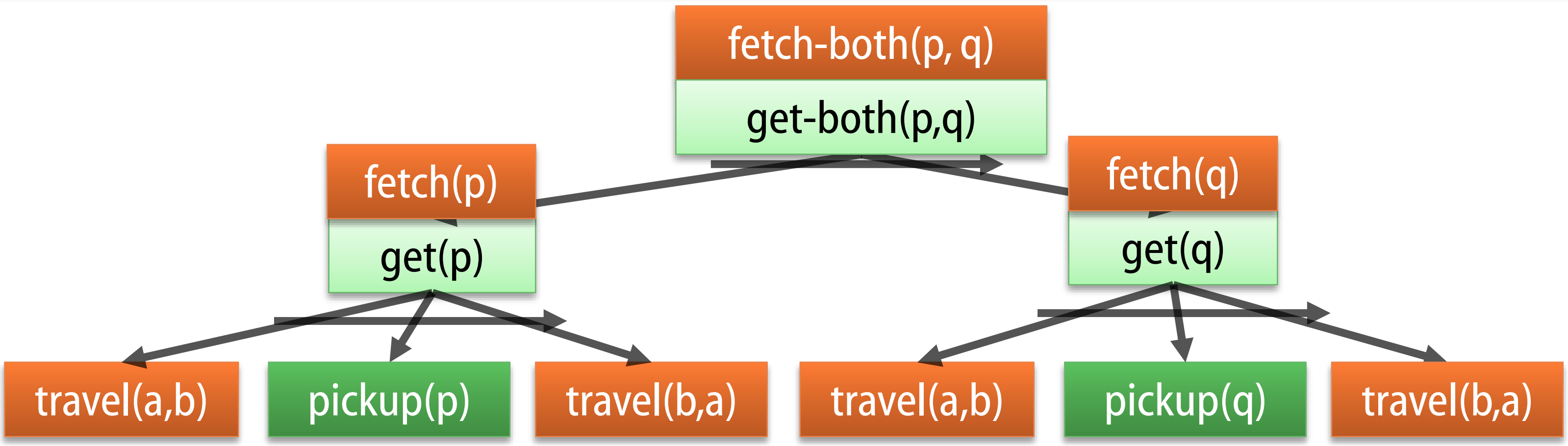
I'm at A, the thing to fetch is at B



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)>

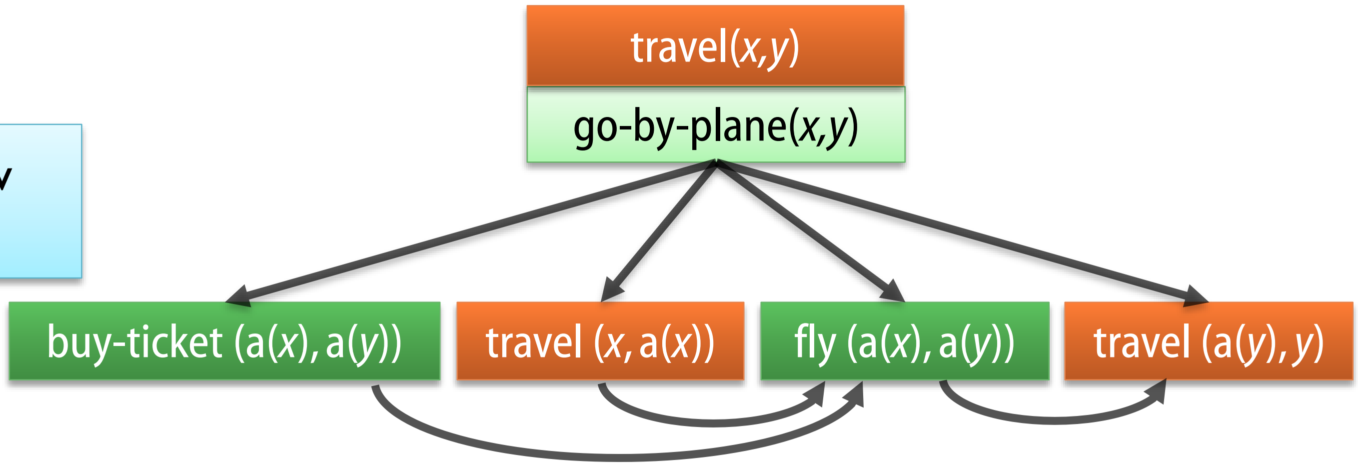
Or: load-all; drive-truck; unload-all

HTN Planning with Partially Ordered Methods

Partially Ordered Methods

- **Partially ordered method**:
  - The subtasks are a **partially ordered** set $\{t_1, \ldots, t_k\}$ – a *network*

No horizontal arrow ordering all tasks



method go-by-plane(x,y)

task: travel(x,y)

precond: long-distance(x,y)

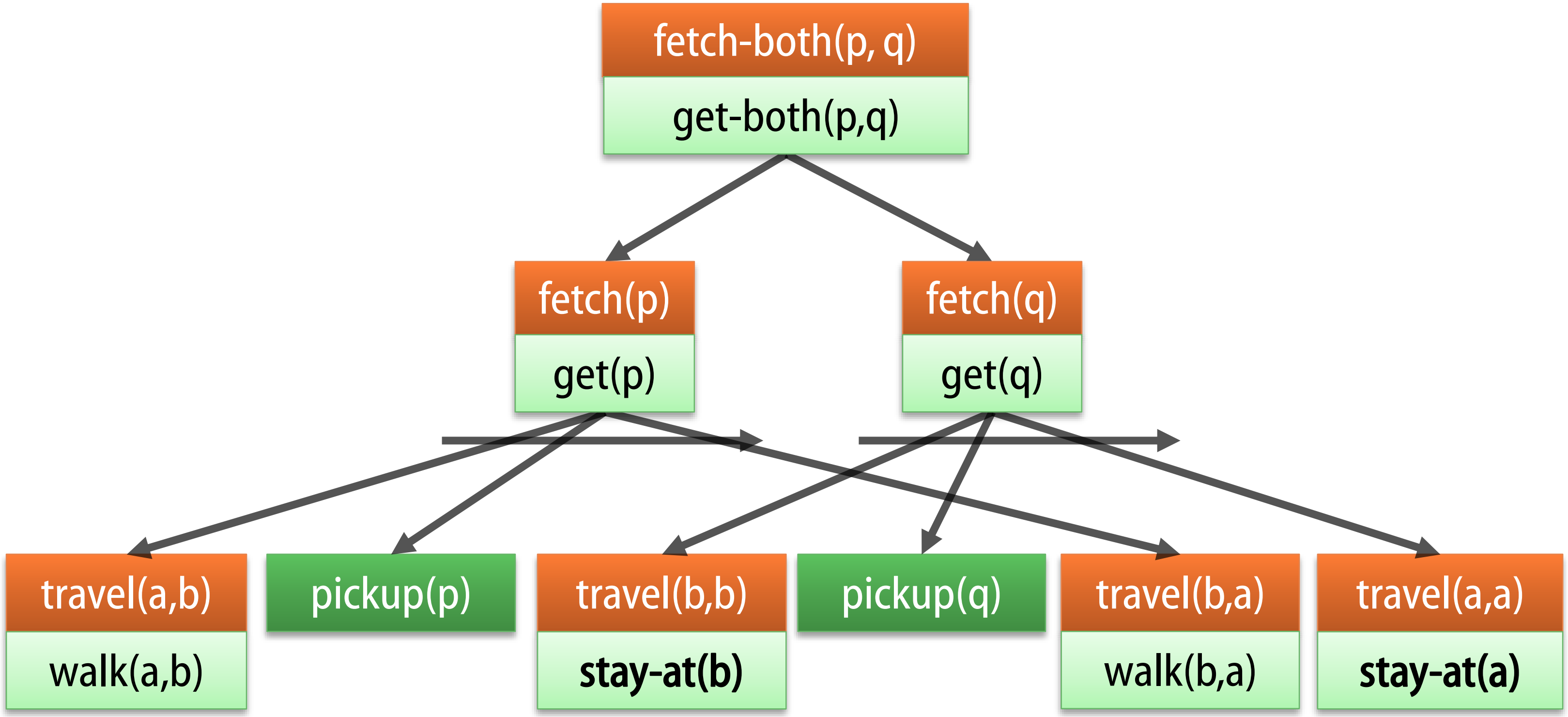
network: u_1=buy-ticket(a(x),a(y)), u_2= travel(x,a(x)), u_3= fly(a(x), a(y))
u_4= travel(a(y),y),
$\{(u_1,u_3), (u_2,u_3), (u_3, u_4)\}$

Precedence: u1 before u3, etc.

Partially Ordered Methods

jonkv@ida

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



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

Conclusion

Conclusion

- 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