Automated Planning

Domain-Configurable Planning: Hierarchical Task Networks

Jonas Kvarnström
Department of Computer and Information Science
Linköping University

Assumptions

jonkv@ida

- Recall the fundamental assumption that we **only** specify
  - Structure: Objects and state variables
  - Initial state and goal
  - **Physical** preconditions and **physical** effects of actions

We only specify what *can* be done

The *planner* should decide what *should* be done

But even the most sophisticated heuristics and domain analysis methods lack our intuitions and background knowledge…

Domain-Configurable Planners

jonkv@ida

How can we make a planner take advantage of what we know?

- Planners taking advantage of additional knowledge can be called:
  - **Knowledge-rich**
  - **Domain-configurable**
  - (Sometimes **incorrectly** called "domain-dependent")

Comparisons (1)

jonkv@ida

More effort

Higher performance

Domain-specific

Must write an entire planner
Can specialize the planner for very high performance

Domain-configurable

High-level (but sometimes complex) domain definition
Can provide more information for high performance

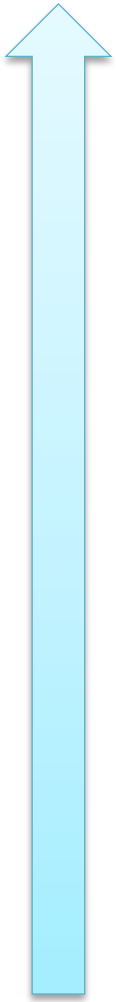
"Domain-independent"

Provide minimal information about actions
Less efficient

Comparisons (2)

jonkv@ida

Larger problem classes
can be handled efficiently



Domain-configurable

Easier to improve expressivity and efficiency
➔ Often practically useful for a larger set of domains!

"Domain-independent"

Should be useful for a wide range of domains

Domain-specific

Only works in a single domain

HTNs: Ideas

jonkv@ida

- Classical Planning vs. 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

jonkv@ida

- A **primitive task** corresponds **directly** to an **action**

stack(A,B)	load(crane1, loc3, cont5, …)
stack(A,B)	load(crane1, loc3, cont5, …)

Primitive tasks: Dark green (in this presentation)

Corresponding actions: Black

  - 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

jonkv@ida

- 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

jonkv@ida

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

Task: travel(*from*, *to*)

Method: go-by-plane(*from*, *to*)
Turns a single travel task into a graph with more details:

- 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

jonkv@ida

In <u>Totally Ordered Simple Task Networks (STN),</u>
each method must specify a <u>sequence</u> of subtasks

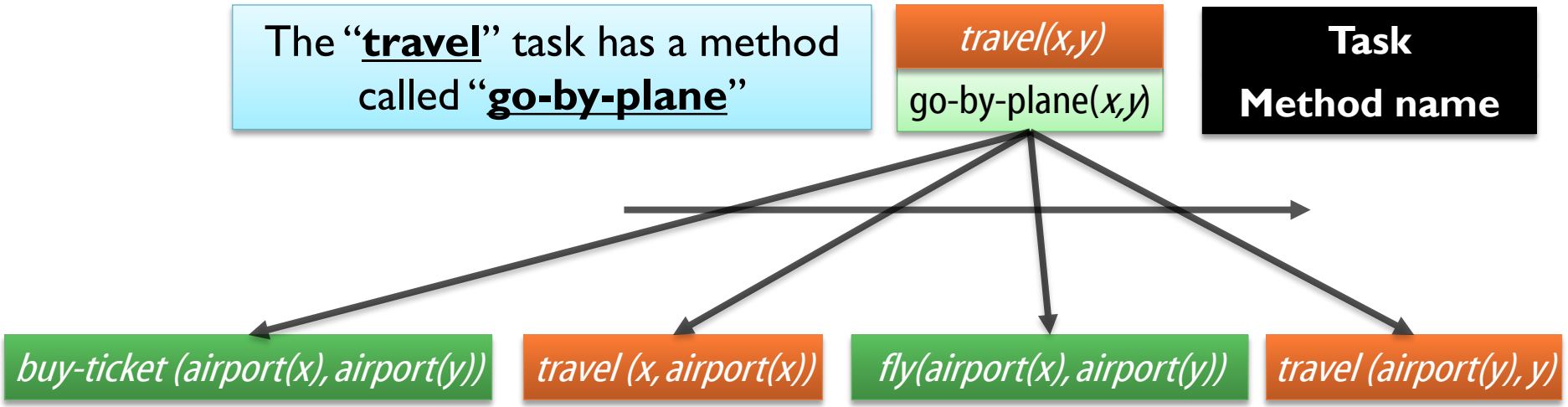
- Can still be modeled as a graph $\langle N, E \rangle$

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

Totally Ordered STNs (2)

jonkv@ida

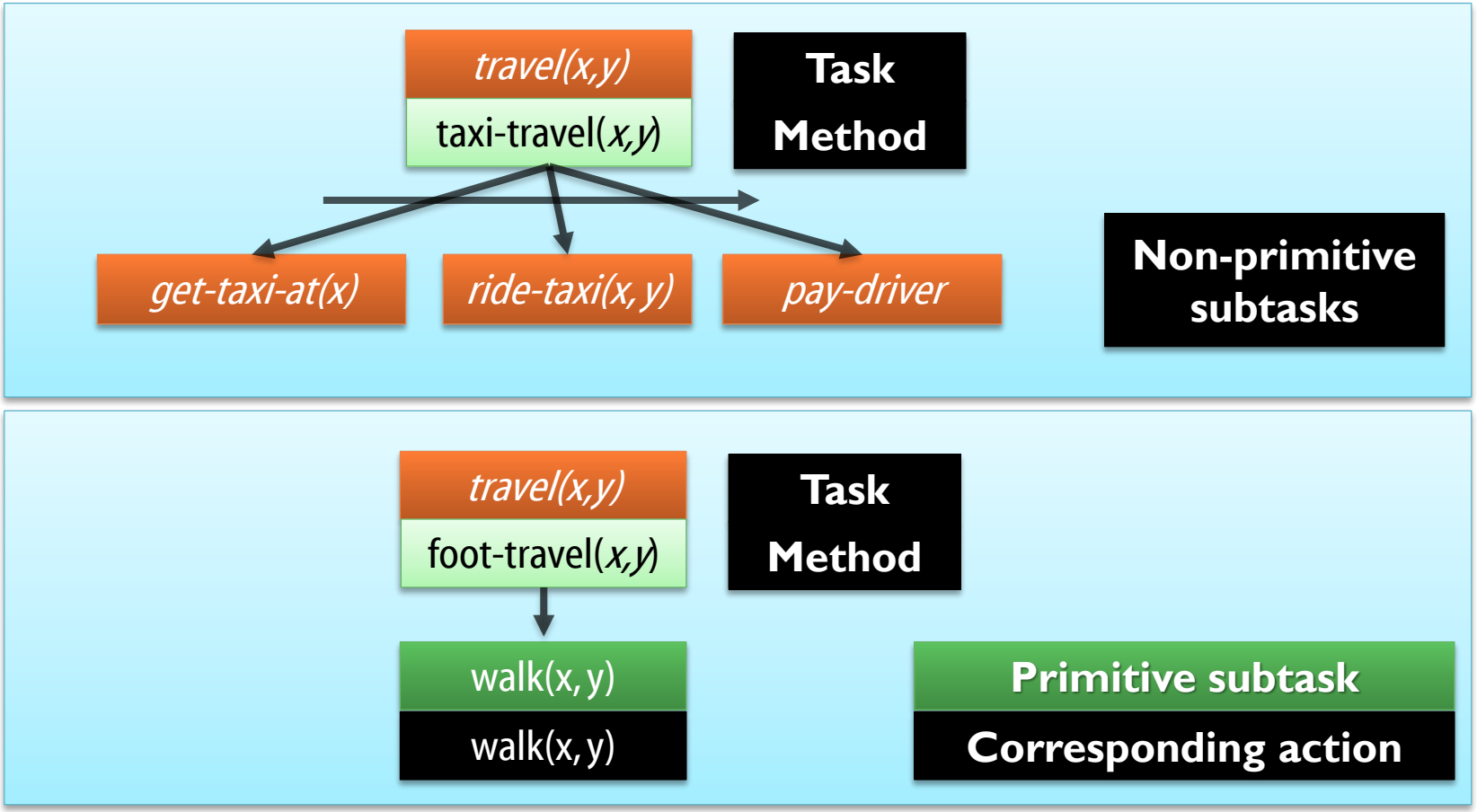
We can **illustrate** the **entire decomposition** in this way
(horizontal arrow ➔ sequence)



Multiple Methods

jonkv@ida

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

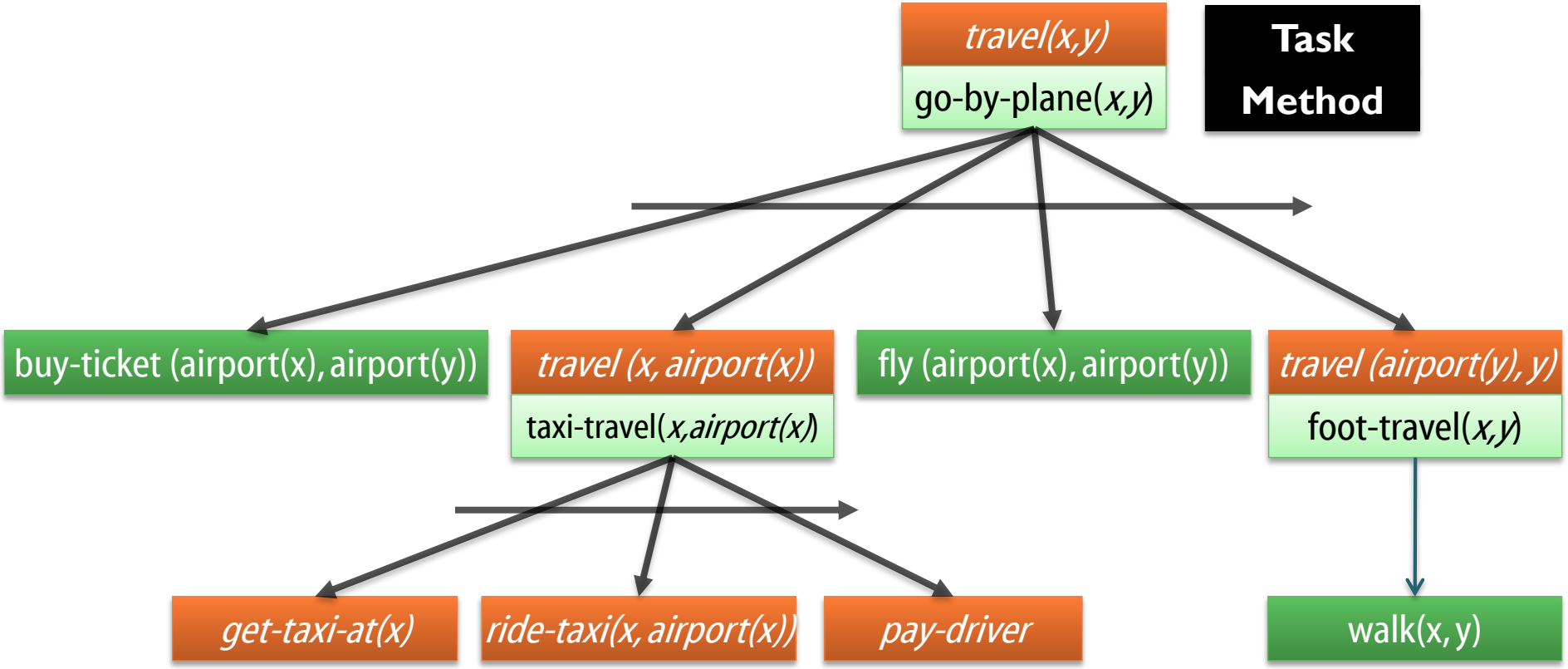


  - …and to determine *parameters* (shown later)

Composition

jonkv@ida

- 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**

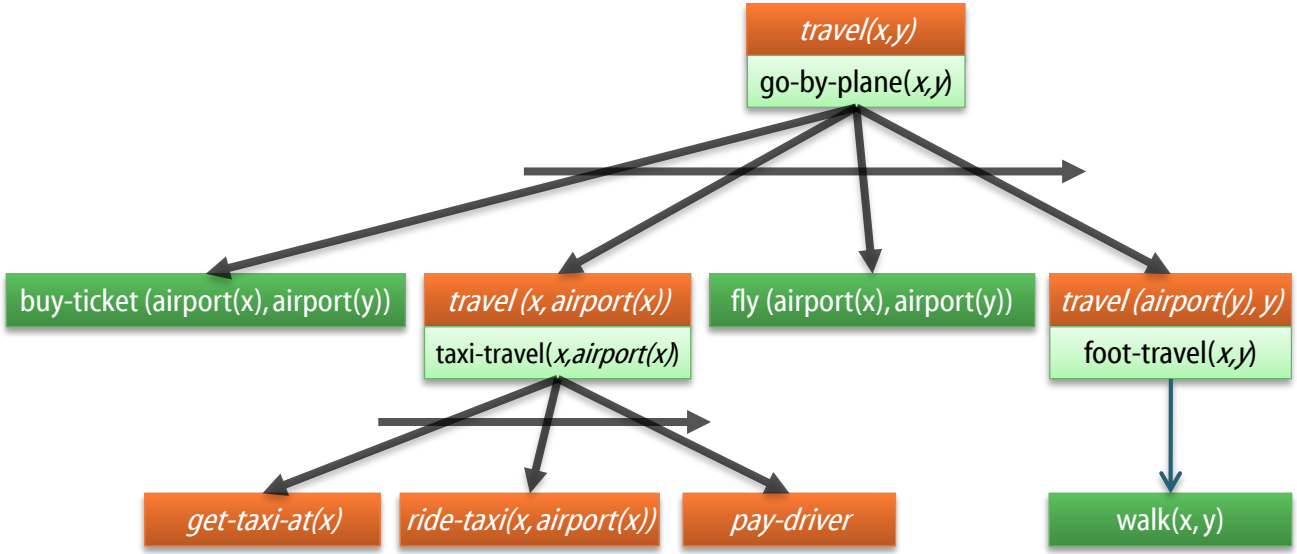
General HTNs: Can have additional constraints to be enforced

- 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)

jonkv@ida

- 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!)

Domains, Problems, Solutions (3)

jonkv@ida

- HTN planning 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

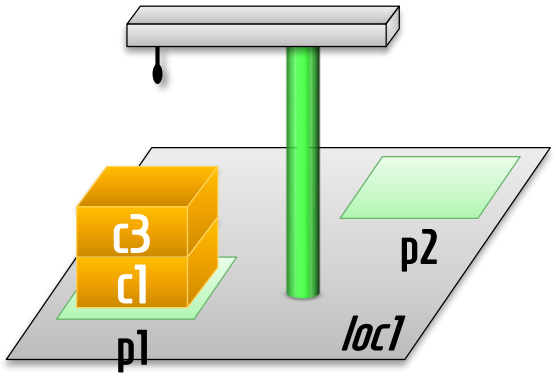
DWR Example: Moving the Topmost Container

A simple "template expansion"

DWR

jonkv@ida

- 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!

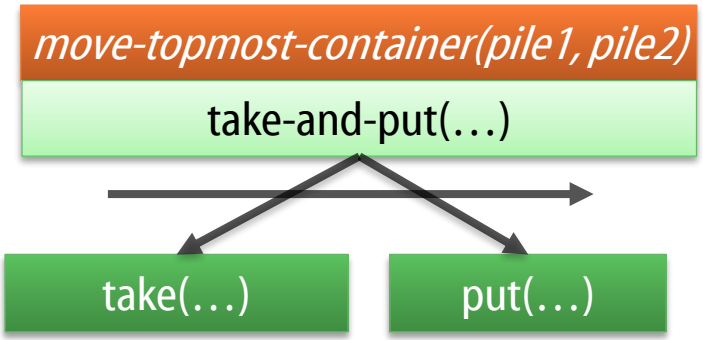
The *precond* adds constraints: crane must be *some* crane in the same loc as the piles, cont must be *the* topmost container of pile1, …

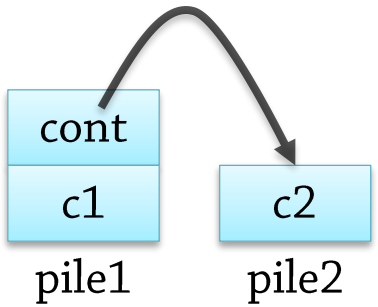
Interpretation:
If you are asked to **move-topmost-container**(pile1, pile2),
check all possible values for **cont, crane, loc, c1, c2** where the preconds are satisfied

Methods (2)

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)
  - **subtasks**: <**take**(crane, loc, cont, c1, pile1),
    put(crane, loc, cont, c2, pile2)>





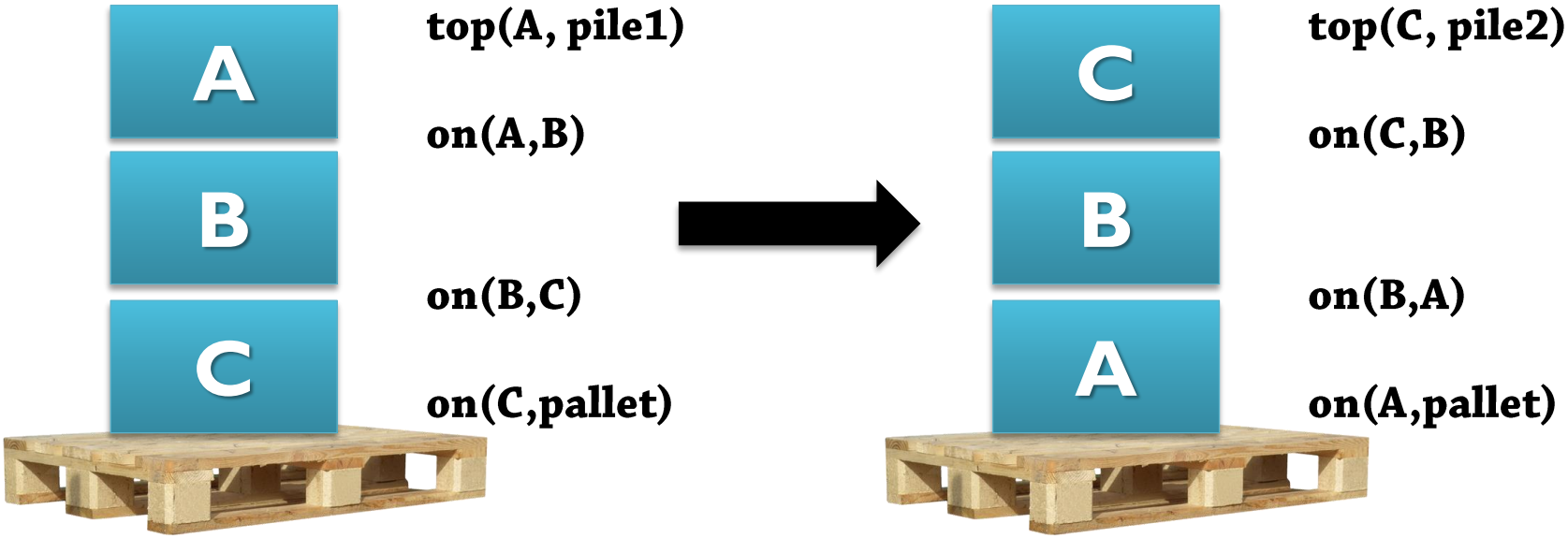
DWR Example: Moving a Stack of Containers

Iteration with no predetermined bound

Moving a Stack of Containers

jonkv@ida

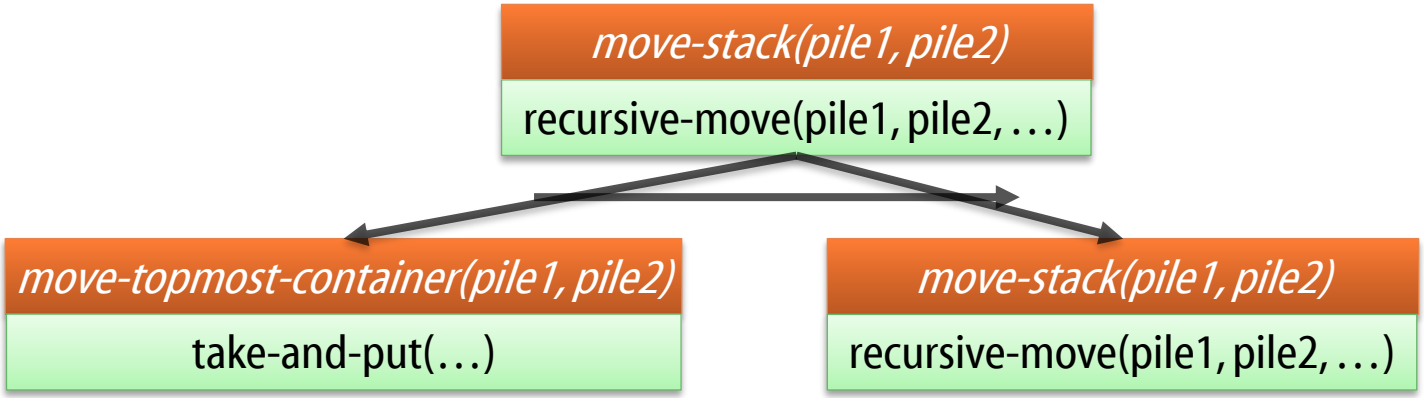
- How can we implement the task **move-stack**(pile1, pile2)?
  - Should move **all** containers in a stack
  - There is no **limit** on how many there might be…



Recursion (1)

jonkv@ida

- 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)

jonkv@ida

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

BW

clear(A)

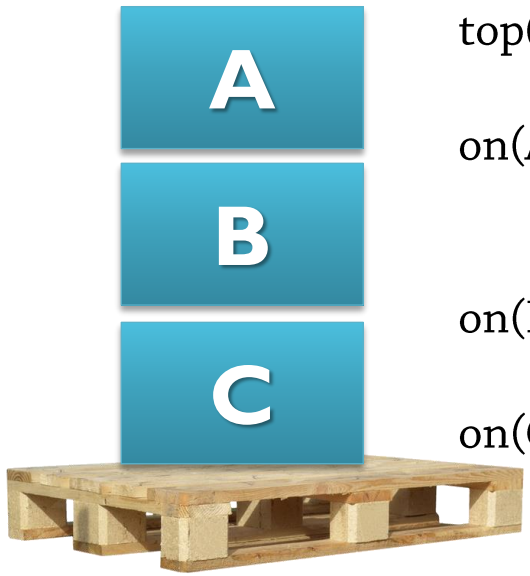
on(A,B)

on(B,C)

ontable(C)

The bottom block
is not "on" anything

DWR



top(A, pile1)

on(A,B)

on(B,C)

on(C,pallet)

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)

jonkv@ida

- To fix this:

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

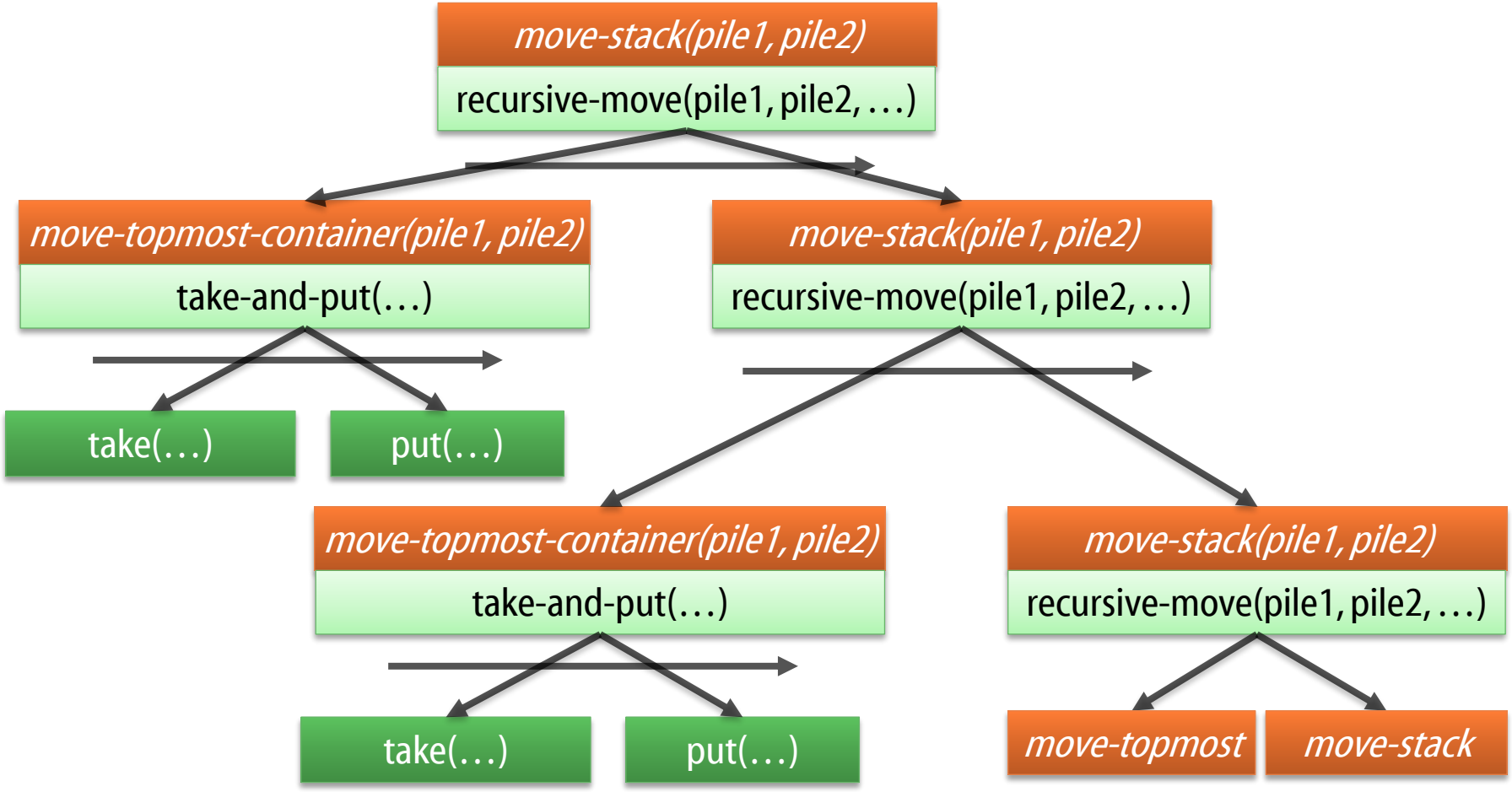
  - **Task**: move-stack(pile1, pile2)
    - **method**: recursive-move(pile1, pile2, ***cont***, ***x***)
    - **precond**: **top(*cont*, pile1), on(*cont*, *x*)**
    - **subtasks**: <move-topmost-container(pile1, pile2), move-stack(pile1, pile2)>

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

Recursion (4)

jonkv@ida

- The planner can now create a structure like this:

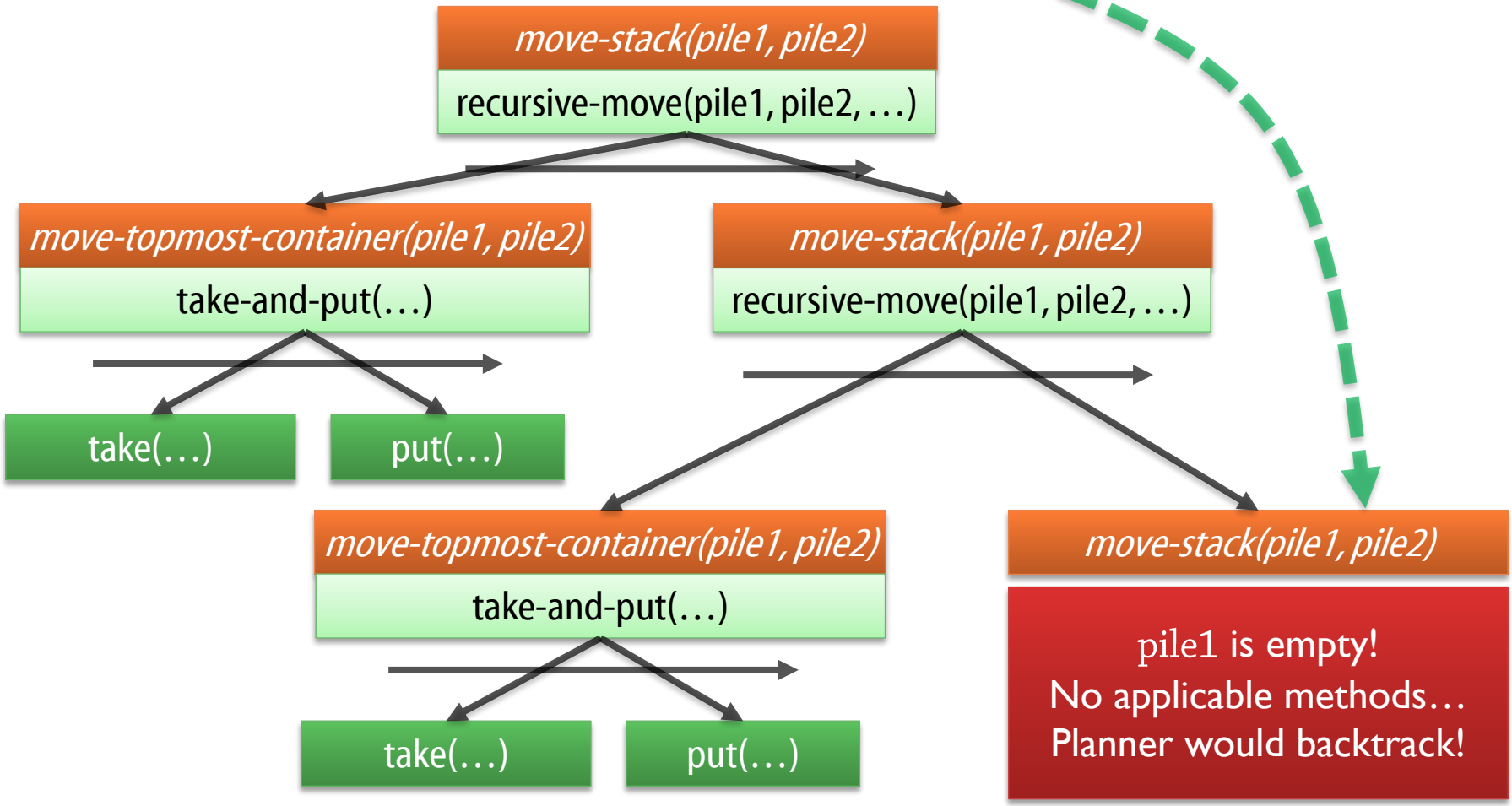


But when will the recursion end?

Recursion (5)

jonkv@ida

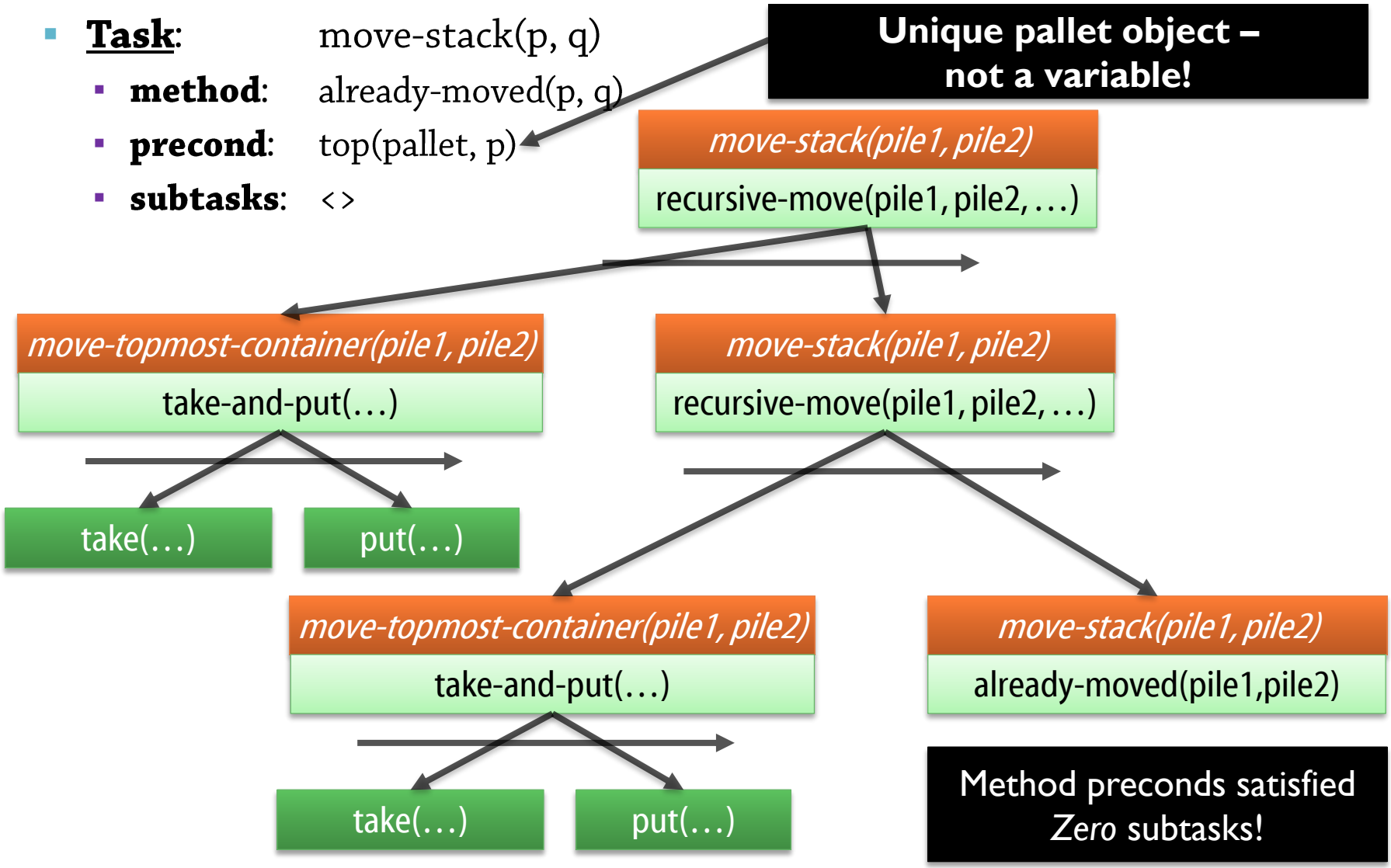
- At some point, **only the pallet** will be left in the stack
  - Then recursive-move will **not be applicable**
  - But we **must** execute **some** form of move-stack!



Recursion (6)

jonkv@ida

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

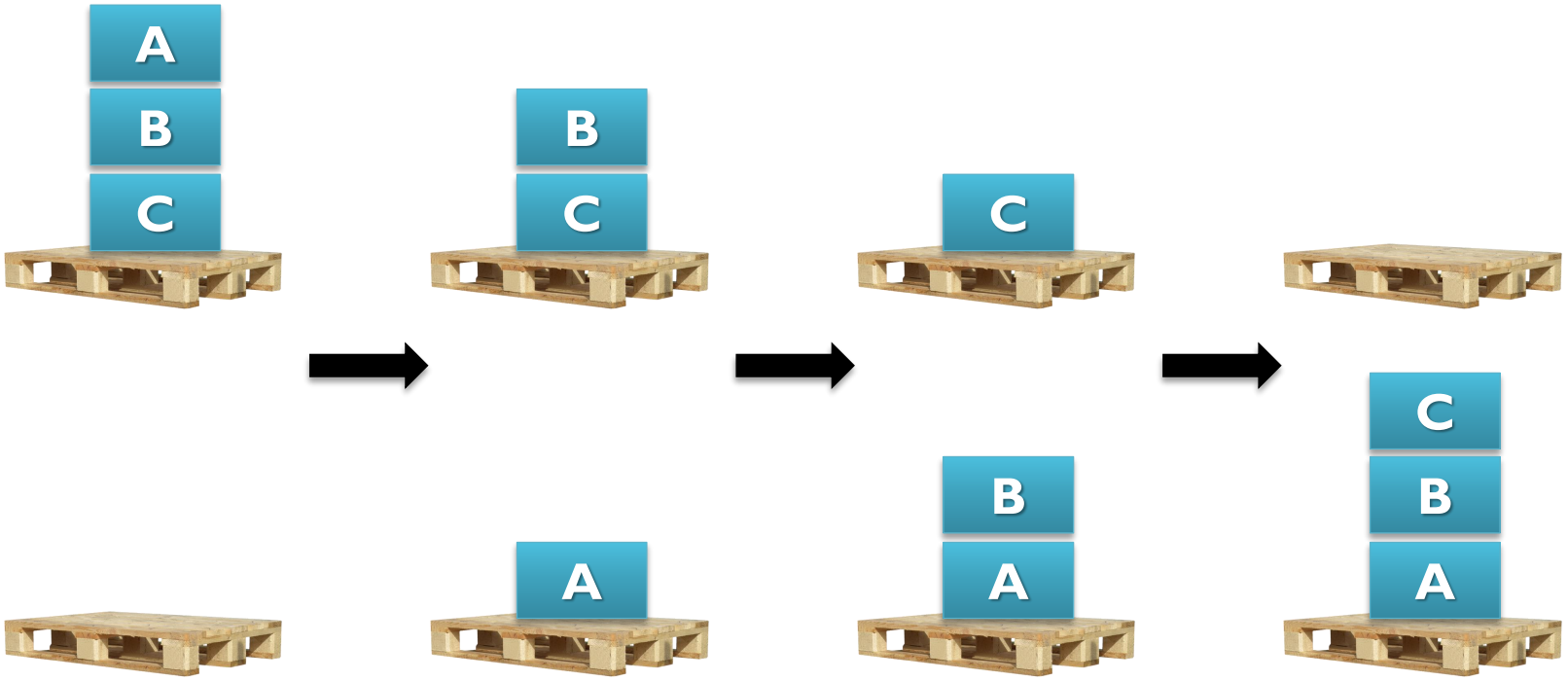


DWR Example:
Moving a stack, in the same order

Ordering (1)

jonkv@ida

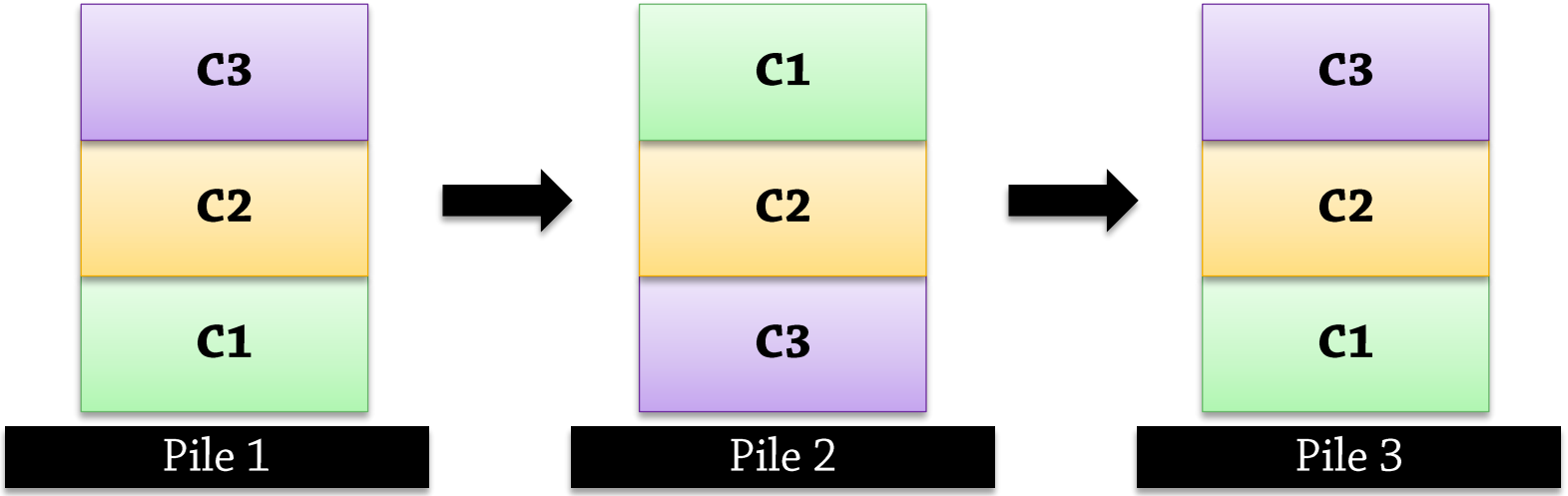
- Using move-stack inverts a stack:



Ordering (2)

jonkv@ida

- To avoid this: Use an intermediate pile



Ordering (3)

jonkv@ida

- 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)>

Unlike classical planning, someone *specifies* the *task* + pile1 and pile2

The planner must choose a matching *method* (”*implementation*”) to use

The planner must choose added method params *pileX* and *loc* to satisfy the precond

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

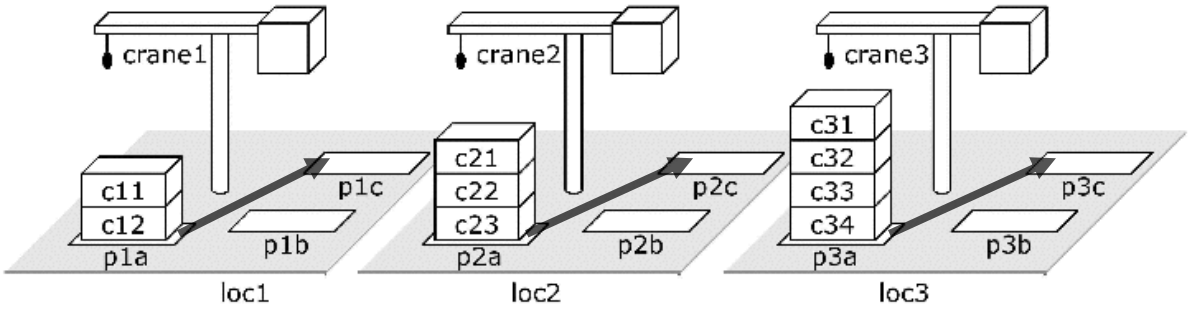
Because the second *move-stack* moves *all* containers from the intermediate pile…

DWR Example: Moving Three Stacks

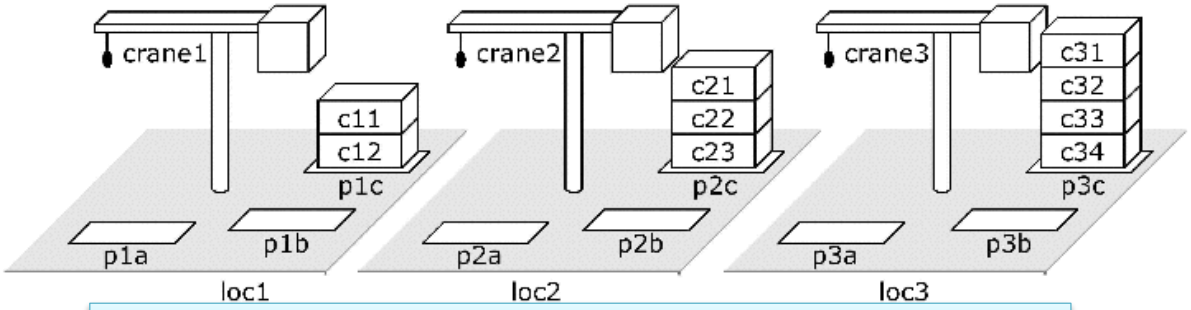
Overall Objective

jonkv@ida

- 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

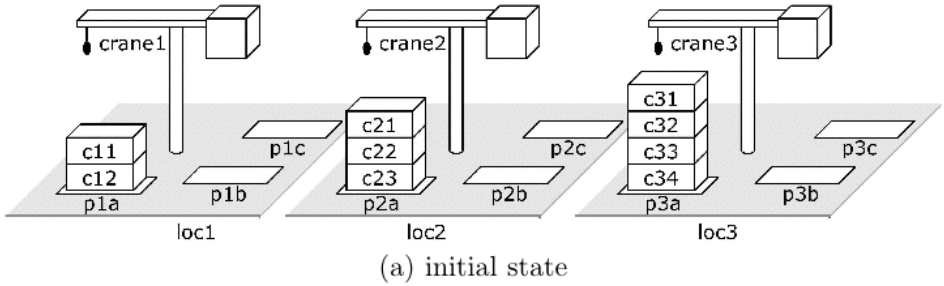
Overall Objective: Defining a Task

jonkv@ida

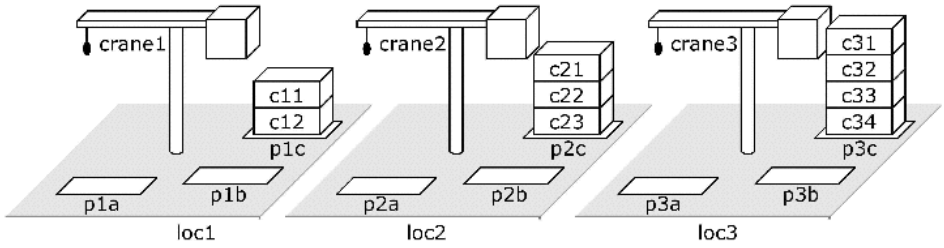
- Define a **task** for this objective
  - **Task:** move-three-stacks()
    - **method**: move-each-twice()
    - **precond**: ; no preconditions apart from the subtasks'
    - **subtasks**: ; move each stack twice:

```
<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

Goal Predicates in HTNs

jonkv@ida

- 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), move-as-necessary> ;; makes should-move… false!
  - **Task:** move-as-necessary()
    - **method**: **all-done**
    - **precond**: not exists pile1, pile2 [should-move-same-order(pile1, pile2)]
    - **subtasks**: < >

Uninformed Planning in HTNs

jonkv@ida

- 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

jonkv@ida

- **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

jonkv@ida

- 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**: <>

One possible search space for Total Order Simple Task Networks:

Total Order Forward Decomposition – TFD

Search Spaces

jonkv@ida

- Need a **search space**

1) A **node structure** defining what information is in a node

2) A way of creating an **initial node** from a problem instance

3) A **successor function** / branching rule returning all successors

4) A **solution criterion**, detecting if a node corresponds to a solution

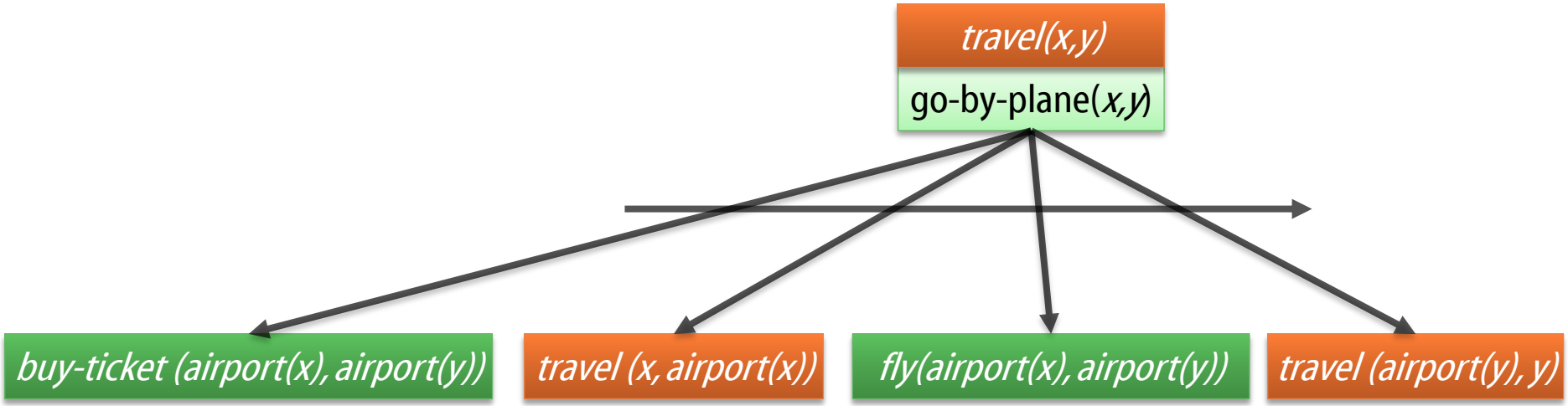
5) A **plan extractor**, telling us which plan a solution node corresponds to

Different alternatives exist!

Total Order?

jonkv@ida

- Basic **assumption**: Total Order Simple Task Networks
  - Any **initial task** is totally ordered
  - Any **decomposition method** is totally ordered



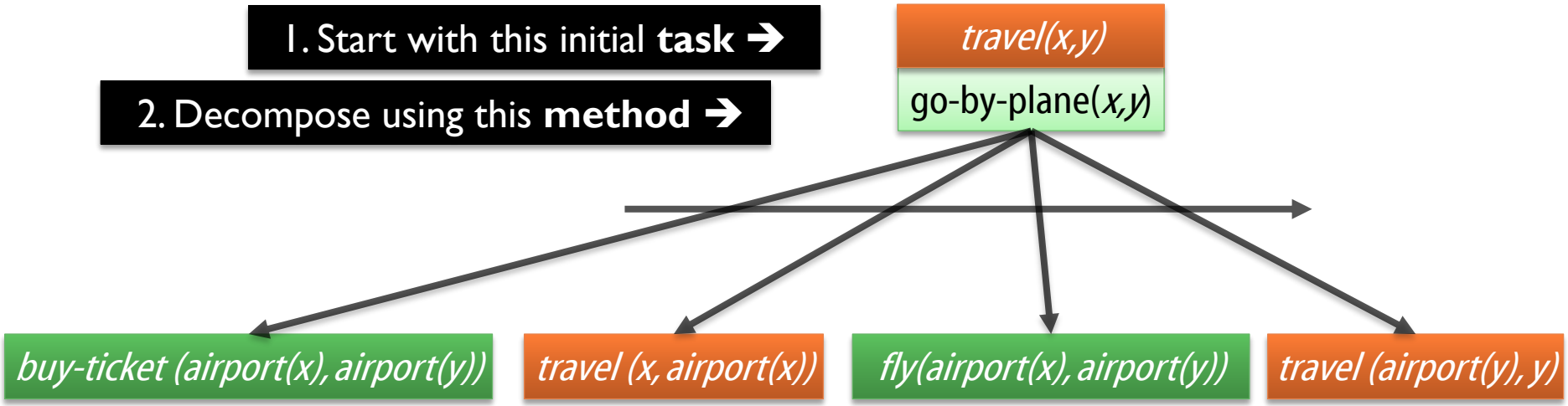
Forward Decomposition?

jonkv@ida

- Different **decomposition orders** are still possible:

1. Start with this initial **task** ➔

2. Decompose using this **method** ➔



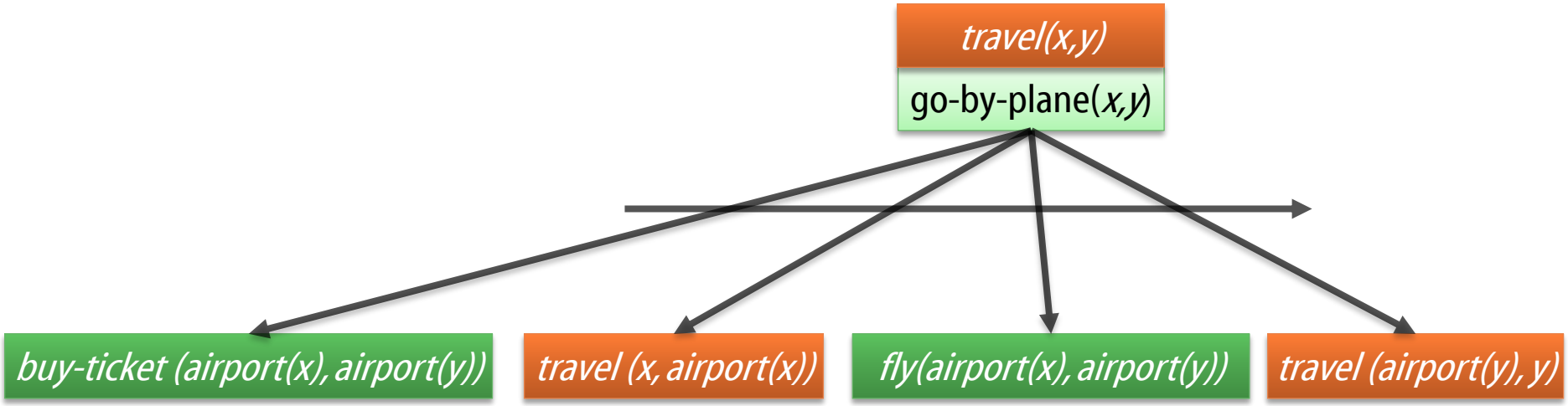
3. Multiple alternatives;
which to decompose next?

Choose what to decompose,
which method to use,
how to parameterize it
➔
Need search!

Forward Decomposition!

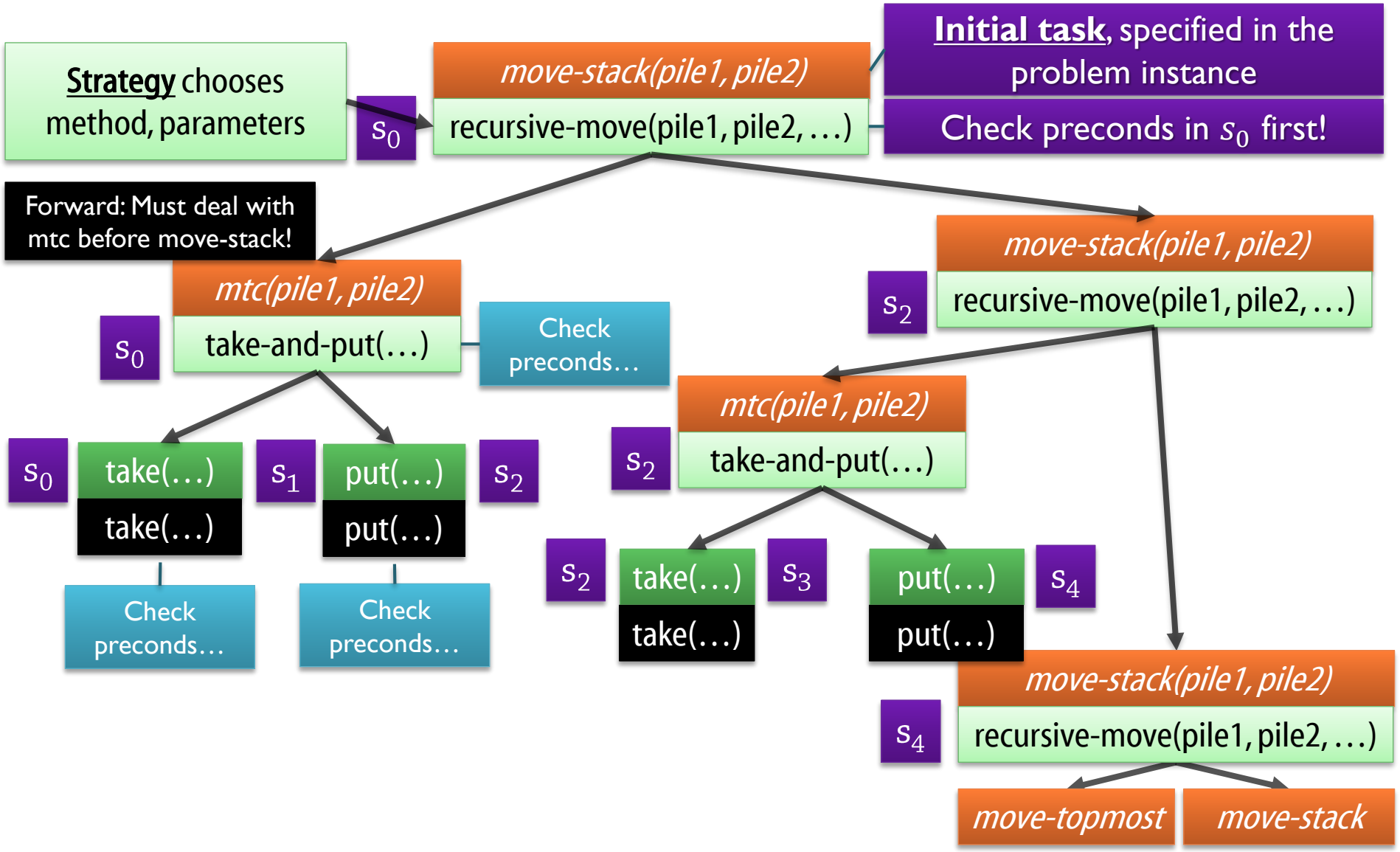
jonkv@ida

- **Forward** decomposition: One of many possibilities
  - Go "depth first, left to right"
  - Like forward state space search:
    - Generates actions in the same order in which they'll be executed
    - ➔ When we decompose a task,
      we know the "current" state of the world!



Total Order Forward Decomposition

jonkv@ida



Total Order Forward Decomposition: The Search Space

TFD Node Structure

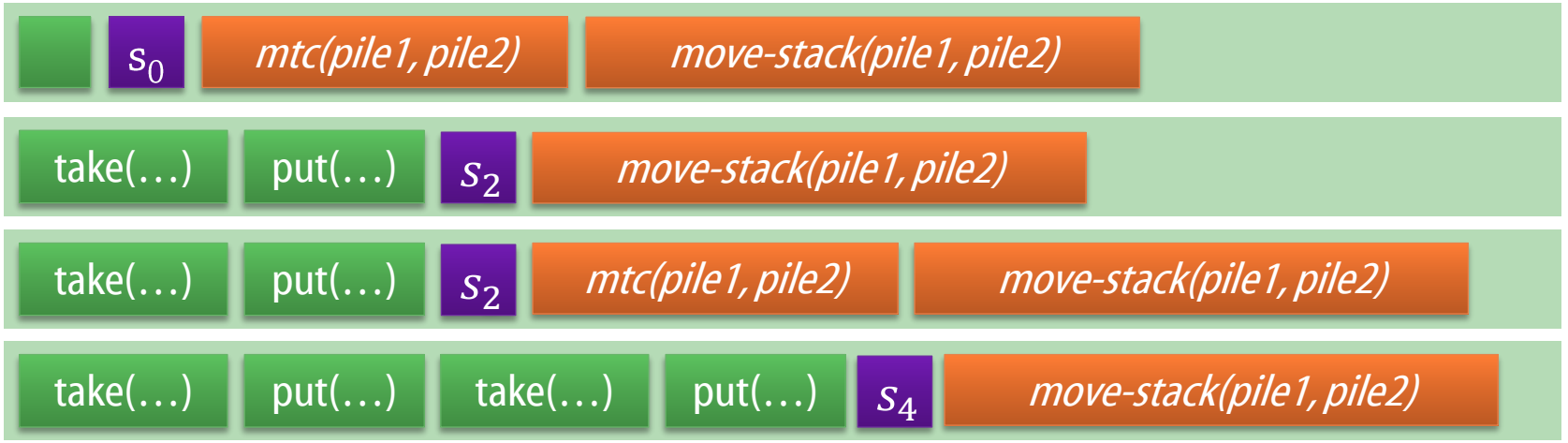
jonkv@ida

- [1] A **node structure** defining what information is in a node
  - Plan so far
  - Current state – *possible due to forward decomposition*
  - Remaining tasks to expand

- [2] A way of creating an initial search node:

No actions so far
Current state s_0
Remaining tasks = the initial tasks from the problem

- Examples: Nodes visited on the previous slide



TFD Successors

jonkv@ida

- **[3]** Successors:
  - We know which task to decompose
  - Find all applicable methods and apply them

- **[4]** Solution test
  - No more tasks ➔ done

- **[5]** Solution extraction
  - The resulting search node *contains* a sequential plan

Total Order Forward Decomposition: The Search Space

Solving Total-Order STN Problems

jonkv@ida

- TFD takes a search node
  - π – a sequence of actions
  - s – the current state
  - <t1,…,tk> – a list of tasks to be achieved **in the specified order**
- We also assume:
  - 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 1: Base case

total-order-forward-decomposition(*problem*) {
    initial-node ← ⟨[], *problem.initialstate*, *problem.initialtask*⟩ **// [2]**
    open ← { *initial-node* }
    while (*open* ≠ ∅) {
        $\langle \pi, s, < t_1, \ldots, t_k > \rangle$ ← **search-strategy-remove-from**(*open*)
        // [6] TFD uses depth first

        // [4] If we have no tasks left to decompose…
        if $k = 0$ **then**
            return π // **[5] plan extraction**

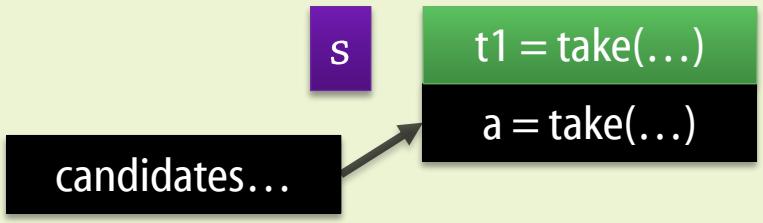
TFD 2: Ground Primitive Tasks

jonkv@ida

total-order-forward-decomposition(*problem*) {
  // initial-node, open
  while (*open* ≠ ∅) {
    $\langle \pi, s, < t_1, \ldots, t_k > \rangle \leftarrow \ldots$
    if $k = 0$ **then return** π

For simplicity: The case where all **tasks to achieve** are **ground**

if (t_1 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
    name(a) = t_1 and
    a is applicable in s }



Waiting in line to be decomposed in the next step

TFD 3: Successors

jonkv@ida

total-order-forward-decomposition(*problem*) {
  // *initial-node, open*
  while (*open* ≠ ∅) {
    $\langle \pi, s, < t_1, \ldots, t_k > \rangle \leftarrow \ldots$
    if $k = 0$ **then return** π

    if (t_1 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
        name(a) = t_1 and
        a is applicable in s }
      for all a ∈ *candidates:*
        π' ← $\pi + a$ **// Add action at the end**
        s' ← $\gamma(s, a)$ **// Apply the action, find the new state**
        rest ← <t2,…,tk>
        open ← *open* ∪ { $\langle \pi', s', rest \rangle$ }

For simplicity: The case where all **tasks to achieve** are **ground**

Total Order Forward Decomposition: Non-Ground Primitive Tasks

TFD 4: Lifted Primitive Tasks

jonkv@ida

total-order-forward-decomposition(*problem*) {
  // initial-node, open
  while (*open* ≠ ∅) {
    ⟨π, *s*, < t_1, … , t_k >⟩ ← …
    if *k* = 0 **then return** π

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

The **plan** will still be **ground**!

if (t_1 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. name(*a*) = **σ(t_1)** and
    a is applicable in s }

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 }

TFD 5: Lifted Primitive Successors

jonkv@ida

total-order-forward-decomposition(*problem*) {

// initial-node, open

while (*open* ≠ ∅) {

⟨π, *s*, < t_1, … , t_k >⟩ ← …

if *k* = 0 **then return** π

if (t_1 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. name(*a*) = **σ(t_1)** and

a is applicable in s }

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

for all (a, σ) ∈ *candidates:*

π' ← $\pi + a$ // Add action at the end

s' ← $\gamma(s, a)$ // Apply the action, find the new state

rest ← σ(<t2,…,tk>) **// Must have the same variable bindings!**

open ← *open* ∪ { ⟨π', s', *rest*⟩ }

(*italics* = variables)

σ(t2) = put(crane1, …)

s: t1 = take(*crane*, loc1, cont2, *cont*, pile8) — t2=put(*crane*, …)

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

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

Total Order Forward Decomposition:
(Non-Ground) Non-Primitive Tasks

TFD 6: Non-Primitive Tasks

jonkv@ida

total-order-forward-decomposition(*problem*) {

// initial-node, open

<u>while</u> (*open* ≠ ∅) {

$\langle \pi, s, < t_1, \ldots, t_k > \rangle \leftarrow \ldots$

<u>if</u> $k = 0$ **then <u>return</u>** π

<u>if</u> (t_1 is primitive) **<u>then …</u>**

else **// t1 is travel(LiU, Resecentrum), for example**

As before, but methods instead of actions

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

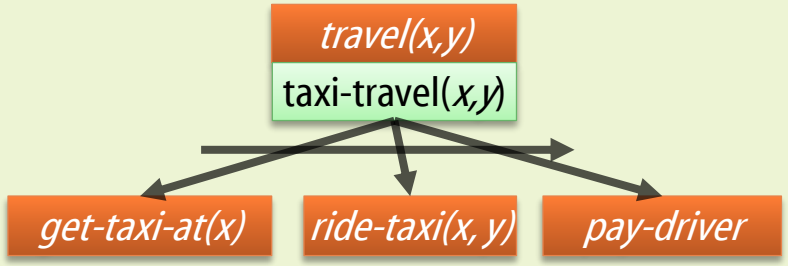
// May have many methods to choose from: taxi-travel, bus-travel, walk, …

ground ← ground instances of methods in M

candidates ← { (m,σ) | m ∈ *ground* and

σ is a substitution s.t. task(m) = $\sigma(t_1)$ and

m is applicable in s } // Methods have preconds!



TFD 6: Non-Primitive Tasks

jonkv@ida

total-order-forward-decomposition(*problem*) {

// initial-node, open

<u>while</u> (*open* ≠ ∅) {

$\langle \pi, s, < t_1, \ldots, t_k > \rangle \leftarrow \ldots$

<u>if</u> $k = 0$ **then <u>return</u>** π

<u>if</u> (t_1 is primitive) **<u>then …</u>**

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(t_1) and

m is applicable in s } // Methods have preconds!

<u>for all</u> $(m, \sigma) \in$ *candidates:*

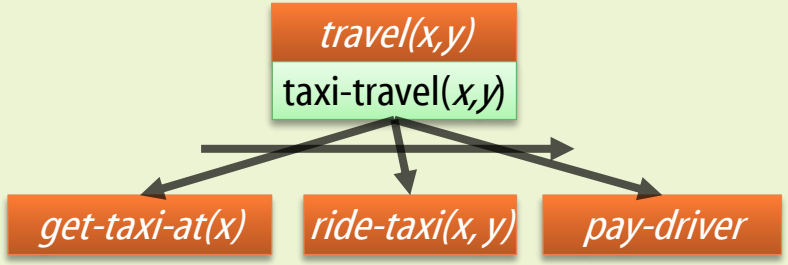
π' ← π **// No action added!**

s' ← s **// No state change!**

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

open ← *open* ∪ { $\langle \pi', s', rest \rangle$ }

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

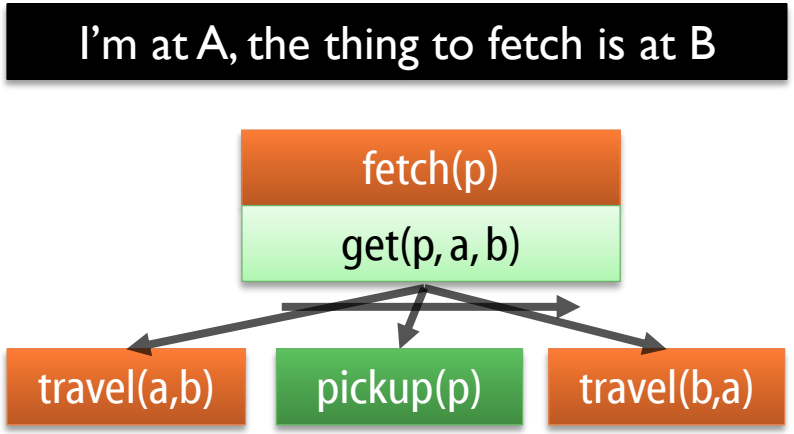


Limitations of Total-Order HTN Planning

Limitation of Ordered-Task Planning

jonkv@ida

- 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)

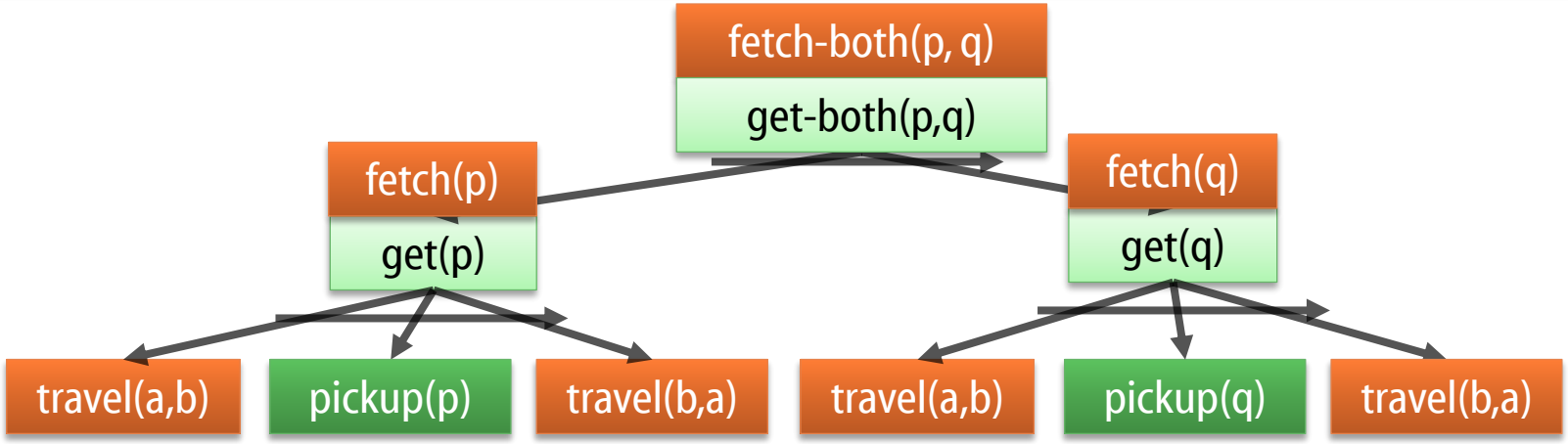


Limitation of Ordered-Task Planning

jonkv@ida

- 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

jonkv@ida

- 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

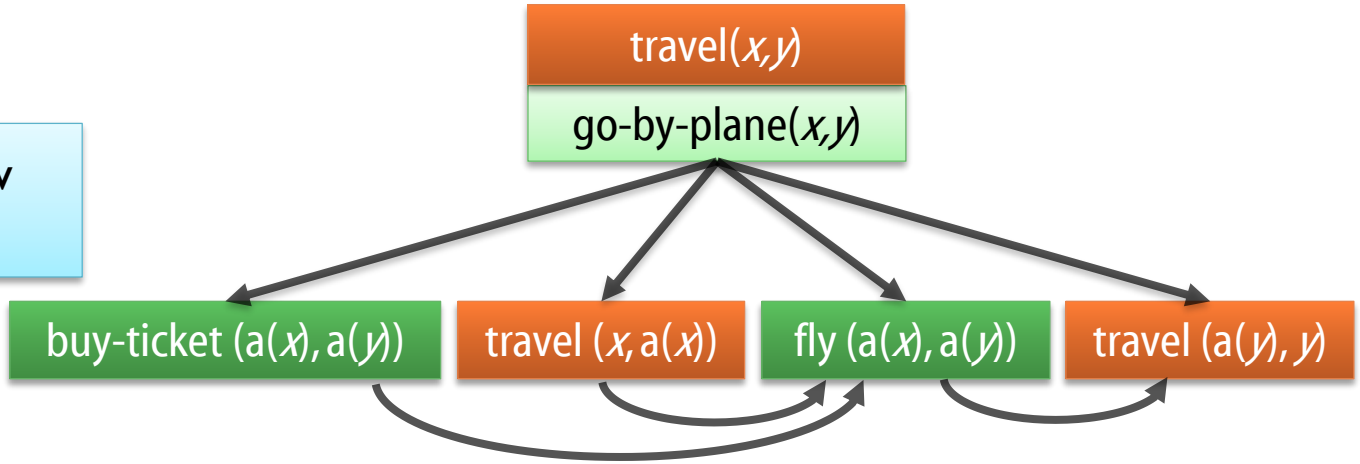
HTN Planning with Partially Ordered Methods

Partially Ordered Methods

jonkv@ida

- **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*)

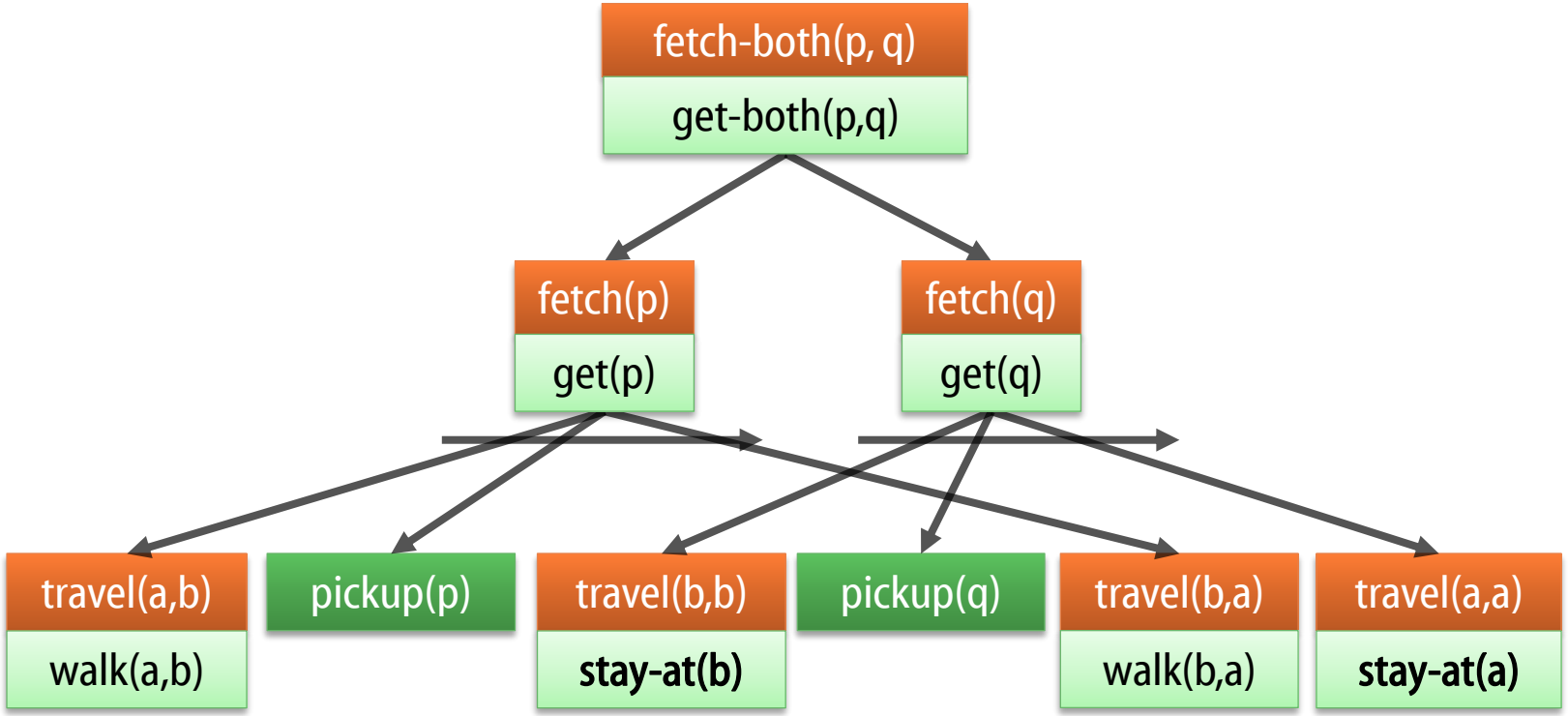
network: u_1=buy-ticket($a(x)$,$a(y)$), u_2= travel(x,$a(x)$), u_3= fly($a(x)$, $a(y)$)
u_4= travel($a(y)$,y),
$\{(u_1,u_3), (u_2,u_3), (u_3,u_4)\}$

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