Course "Automated Planning: Theory and Practice"
Chapter 10: Domain-Configurable Planning: Hierarchical Task Networks

Teacher:	Marco Roveri - marco.roveri@unitn.it
M.S. Course:	Artificial Intelligence Systems (LM)
A.A.:	2023-2024
Where:	DISI, University of Trento
URL:	https://bit.ly/3zOkGk8

Last updated: Monday 23rd October, 2023

Terms of Use and Copyright

The material (text, figures) in these slides is authored by Jonas Kvarnström and Marco Roveri.

Assumptions

- The fundamental assumptions we considered so fare are:
  - We only specify: Objects and state variables
  - We only specify: initial state and goal
  - Physical preconditions and effects of actions

$$\Downarrow$$

- We only specify what can be done!
- The planner should decide what should be done!

$$\Downarrow$$

- But... even the most sophisticate heuristics and domain analysis methods lack human intuitions and background knowledge...

Domain-Configurable Planners

- 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"

Comparison

More Effort

Higher Performance

Domain-Specific

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

Domain-Configurable

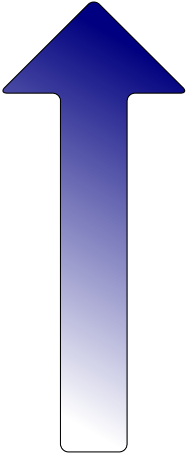
High-level (but sometimes complex) domain definition
Can provide more information for high performance

Domain-Independent

Provide minimal information about actions
Less efficient

Comparison (cont.)

Larger problem classes
can be handled efficiently



Domain-Configurable

Easier to improve expressivity and efficiency
$\Longrightarrow$ 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

Hierarchical-Task Networks: Intuition

Classical Planning

- Objective is to achieve a goal

```
{(at TimeSqaure)}
{(on A B), (on C D)}
...
```

$\Longrightarrow$

Hierarchical Task Networks

- Objective is to perform a task

```
{(travel-to TimeSqaure)}
{(place-blocks-correctly)}
...
```

- Find any sequence of actions that achieves the goal

$\Longrightarrow$

- Use "templates" to incrementally refine the task until *primitive* actions are reached!

```
(travel-to TimeSqaure)
        ⇓
  (taxi-to airport);
  (fly-to JFK); ...
```

Provides guidance but still requires planning!

Terminology: Primitive Task

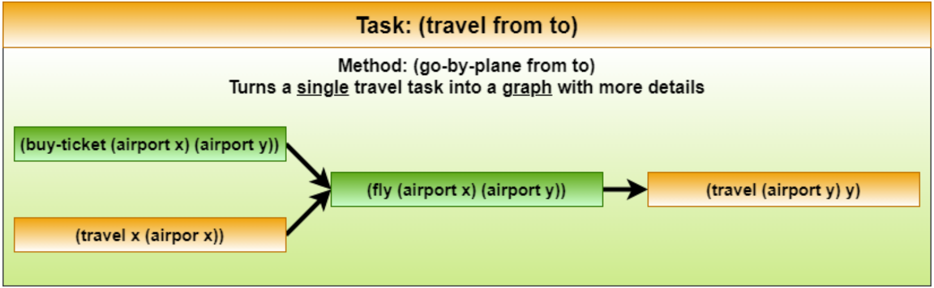
- A primitive task corresponds directly to an action
  - As in classical planning, what is primitive depends on:
    - The execution system!
    - How detailed you want your plans to be!
  - Example:
    - For you the `(fly from to)` may be a primitive task
    - For the pilot, it may be further decomposed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

Terminology: Non-Primitive Task

- A non-primitive task
  - Cannot be directly executed
  - Must be decomposed into 1 or more sub-tasks
  - Example:
    - `(put-all-blocks-in-place)`
    - `(make-tower A B C D E)`
    - `(move-stack-of-blocks x y)`

Terminology: Method

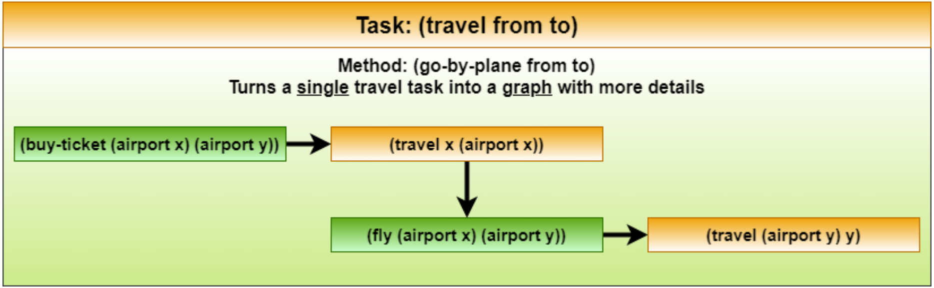
- A method specifies *one way* to decompose a non-primitive task into sub-tasks.



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

Totally Ordered Simple Task Networks

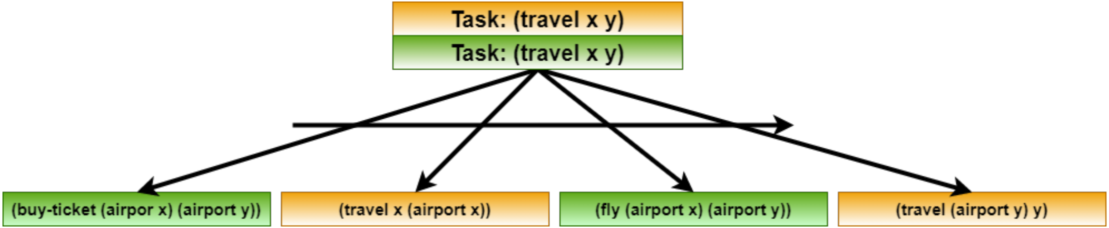
- In totally ordered simple task networks each method must specify a sequence of sub-tasks!



- Alternatively: A sequence $\langle t_1, ..., t_k \rangle$
  - ⟨ `(buy-ticket (airport x) (airport y)),`
    `(travel x (airport x)),`
    `(fly (airport x) (airport y)),`
    `(travel (airport y) y)` ⟩

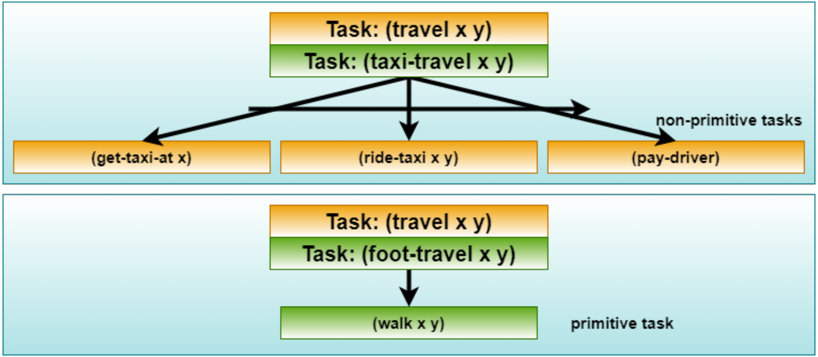
Totally Ordered Simple Task Networks

- We illustrate the entire decomposition using an horizontal arrow $\longrightarrow$ to represent the sequence!



Multiple Methods

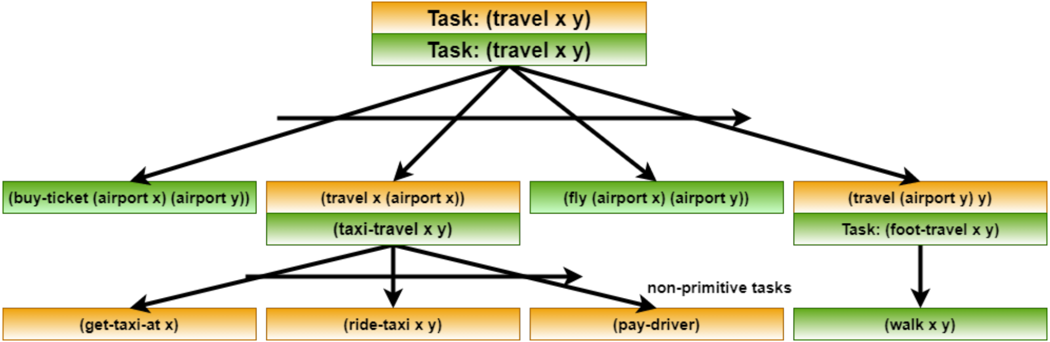
- A non-primitive task can have many methods



  - $\Longrightarrow$ You still need to search to determine which method to use!
  - $\Longrightarrow$... and to determine the *parameters* (discussed later)!

Composition

- A Hierarchical Task Network plan:
  - Hierarchical
  - Consists of tasks
  - Based on graphs ≈ networks



Domains, Problems, Solutions

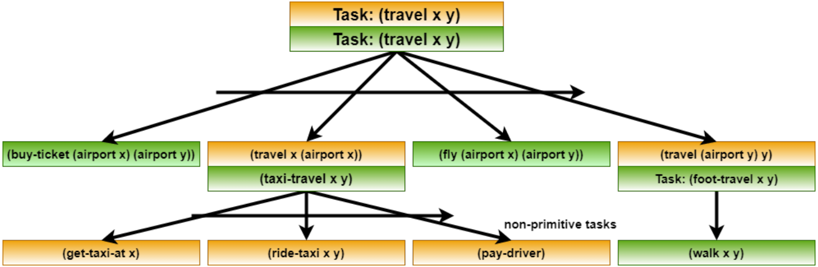
- A Simple Task Network planning domain specifies:
  - A set of tasks
  - A set of operators used as primitive tasks
  - A set of methods

- A Simple Task Network problem instance specifies:
  - A Simple Task Network planning domain
  - An initial state
  - An initial task network which shall be ground (no variables)
    - A total order Simple Task Network example:
      ⟨`(travel home work)`, `(do-work)`, `(travel work home)`⟩

General Hierarchical Task Networks can have additional constraints to be enforced!

Domains, Problems, Solutions (cont.)

- 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



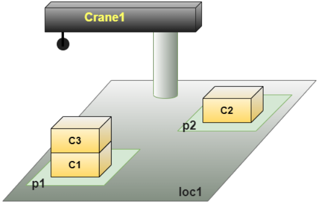
- Totally ordered $\Longrightarrow$ yields an action sequence
  - If this is executable: A solution
  - No goals to check – they are implicit in the method structure!

Domains, Problems, Solutions (cont.)

- Hierarchical Task Network 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!

Dock Worker Robots

- Example tasks:
  - Primitive – All the DWR actions we considered so far
  - Move the topmost container between piles
  - Move the 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

- To move top most container from one pile to another
  - **task** `(move-topmost-container pile1 pile2)`

    The *task* has parameters given from above
  - **method** `(take-and-put cont crane loc pile1 pile2 c1 c2)`

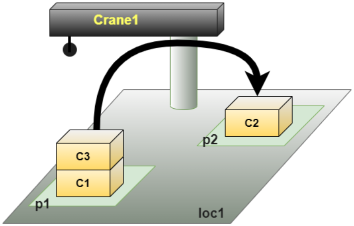
    A *method* can have additional parameters, whose values are chosen by the planner – as in classical planning!
  - **precond:** `(attached pile1 loc) (attached pile2 loc) (belong crane loc) (top cont pile1) (on cont c1) (top c2 pile2)`

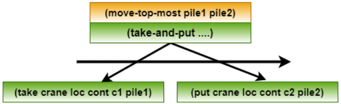
    The *precond* adds constraints: `crane` must be *some* crane in the same `loc` as the piles, `cont` must be the top most container of `pile1`, ...

Intepretation: If you're asked to `(move-topmost-container pile1 pile2)`, check all possible values for `cont`, `crane`, `loc`, `c1`, `c2` where the preconditions are satisfied!

Methods

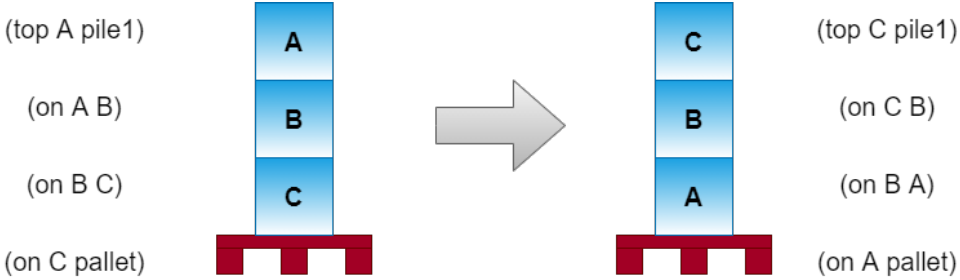
- To move top most container from one pile to another
  - **task**
    `(move-topmost-container pile1 pile2)`
  - **method**
    ```
    (take-and-put cont crane loc
                  pile1 pile2 c1 c2)
    ```
  - **precond:** `(attached pile1 loc)`
    `(attached pile2 loc)`
    `(belong crane loc)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





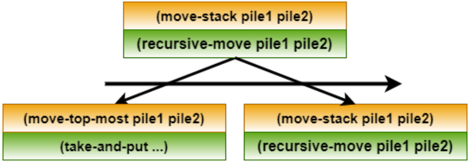
Moving a Stack of Containers

- Ho can we implement the `(move-stack pile1 pile2)`?
  - Should we move all containers in a stack?
  - There is no limit on how many there might be...



Recursion

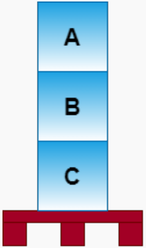
- We need a loop with a termination condition
  - Hierarchical Task Network planning allows recursion
    - Move the topmost container (we know how to do that!)
    - Then move the rest
  - First attempt:
    - **task**: `(move-stack pile1 pile2)`
    - **method**: `(recursive-move pile1 pile2)`
    - **precond**: `True`
    - **subtasks**: ⟨ `(move-topmost-container pile1 pile2)`, `(recursive-move pile1 pile2)` ⟩



Recursion (cont.)

- Let's consider the BW and the DWR "pile models"...

BW



(top A pile1)

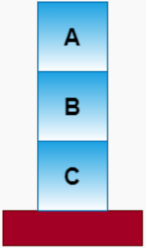
(on A B)

(on B C)

(on C pallet)

The bottom block is not "on' anything!

DWR



(clear A)

(on A B)

(on B C)

(ontable C)

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

What if the pallet is "topmost"? We do not want to move it!

Recursion (cont.)

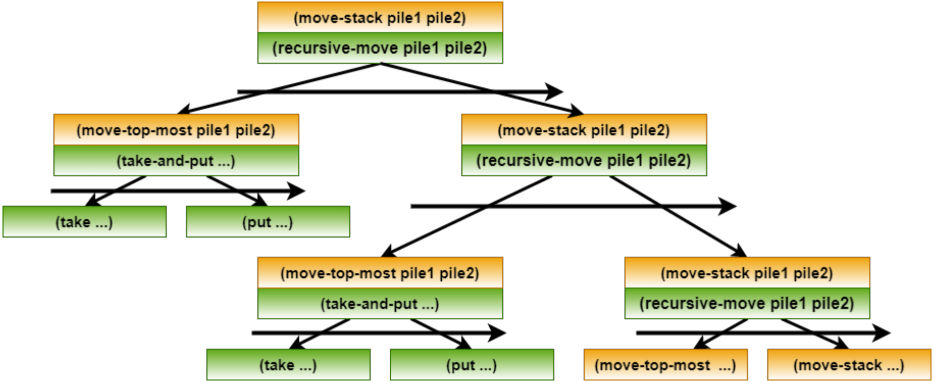
- To fix this
  - **task**: `(move-stack pile1 pile2)`
  - **method**: `(recursive-move pile1 pile2 cont x)`
  - **precond**: `(top cont pile1)` `(on cont x)`
  - **subtasks**: ⟨ `(move-topmost-container pile1 pile2)` `(move-stack pile1 pile2)` ⟩

`cont` is on top of something (i.e. `x`), so `cont` can't be the `pallet`!

We added two additional method parameters (`cont` `x`) – "non-natural", as in "ordinary" planning $\Longrightarrow$ does not give the planner a real choice!

Recursion (cont.)

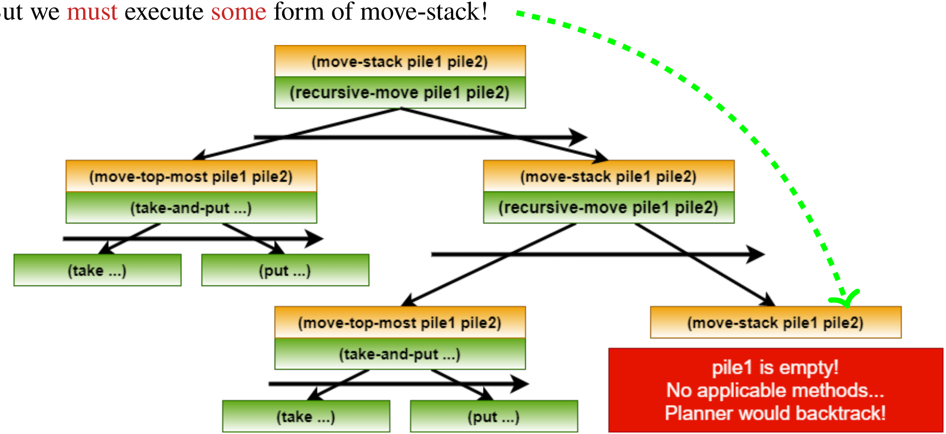
- The planner can create a structure like this...



- ... but when will the recursion end?

Recursion (cont.)

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

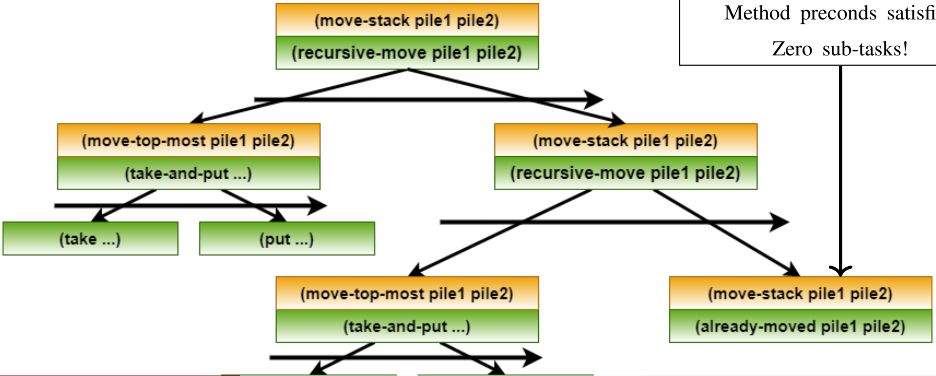


Recursion (cont.)

- We need a method that terminates the recursion
  - **task**: `(move-stack pile1 pile2)`
  - **method**: `(already-moved pile1 pile2)`
  - **precond**: `(top pallet pile1)`
  - **subtasks**: $\langle\,\rangle$

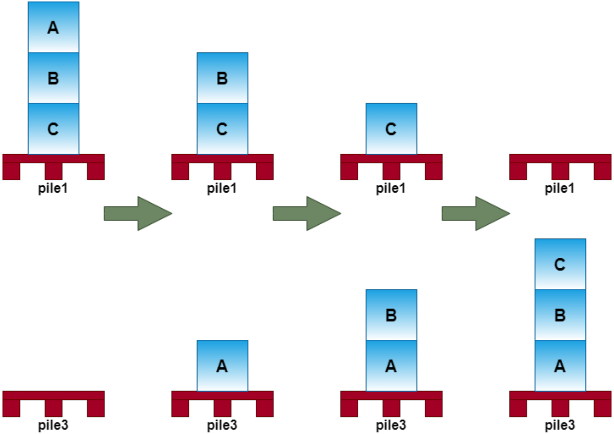
Unique `pallet` object
not a variable

Method preconds satisfied
Zero sub-tasks!



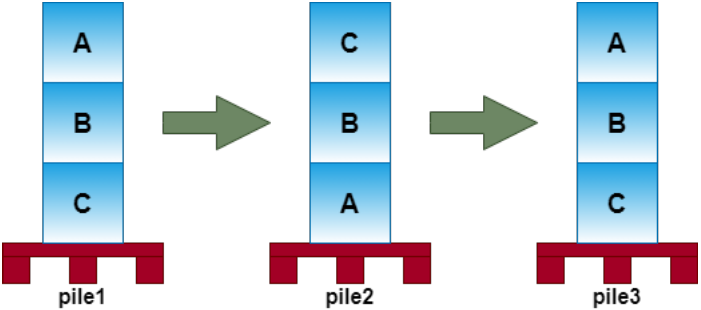
Ordering

- Using move-stack inverts a stack!



Ordering (cont.)

- To avoid this: use intermediate pile



Ordering (cont.)

- 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 `loc`
    ...
  - **subtasks**: ⟨ `(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

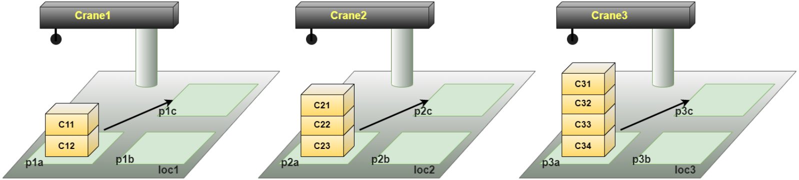
The planner must choose added method params `pileX` `loc` to satisfy the precond!

Why does `pileX` have to be empty initially?

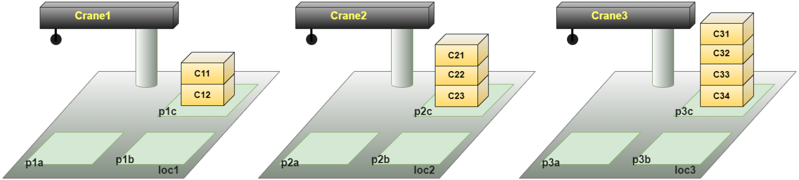
Because the second move-stack moves all containers from the intermediate `pileX` to destination `pile2`!

Overall objective

- Moving three entire stacks of containers preserving order!



Initial state, with three locations, three piles to move

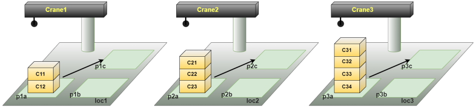


Corresponding objective: all piles moved!

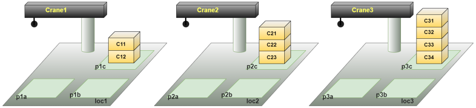
Overall objective: Defining a task

- Define a task for this objective!
  - **task**: `(move-three-stacks)`
  - **method**: `(move-each-twice)`
  - **precond**: // no preconditions apart from the sub-tasks'
  - **subtasks**: ⟨ `(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*



Initial state, with three locations, three piles to move



Corresponding objective: all piles moved!

Goal Predicates in Hierarchical-Task Networks

- Here the entire objective is encoded in the initial network `(move-three-stacks)`
  - $\Longrightarrow$ ⟨ `(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 Hierarchical-Task Networks

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

    Planner chooses all parameters!
  - Repeating
    - Task `(achieve-goals)` $\Longrightarrow$ $\langle$ `(do-something)`, `(achieve-goals)` $\rangle$
  - Ending
    - Task `(achieve-goals)` $\Longrightarrow$ $\langle\,\rangle$, with precond: entire goal is satisfied!

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

DELIVERY: FIRST VARIATION

- Delivery:
  - A single truck
  - Pick-up a package, drive to 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: Second Variation

- Alternative: Two alternative methods `deliver`

  - **task**: `(deliver package dest)`
  - **method**: `(move-by-truck-1 package packageloc truckloc dest)`
  - **precond**: `(at truck truckloc) (at package packageloc) (= truckloc packageloc)`
  - **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) (!= truckloc packageloc)`
  - **subtasks**: ⟨ `(driveto packageloc), (load package),`
    `(driveto dest), (unload package)` ⟩

Do we really have to repeat the entire task?
Many "conditional" sub-tasks $\Longrightarrow$ combinatorial explosion!

Delivery: Second Variation

- Make the choice in the sub-task instead!
  - **task**: `(deliver package dest)`
  - **method**: `(move-by-truck-3 package packageloc truckloc dest)`
  - **precond**: `(at truck truckloc) (at package packageloc) (= truckloc packageloc)`
  - **subtasks**: ⟨ `(be-at packageloc), (load package), (be-at dest), (unload package)` ⟩

  - **task**: `(be-at loc)`
  - **method**: `(drive loc)`
  - **precond**: `(not (at truck loc))`
  - **subtasks**: ⟨ `(driveto loc)` ⟩

  - **task**: `(be-at loc)`
  - **method**: `(already-there)`
  - **precond**: `(at truck loc)`
  - **subtasks**: ⟨ ⟩

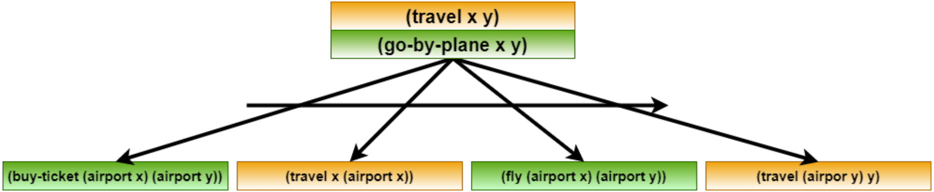
Search Spaces

- Need search space
  - 1) A node structure defining what information is in a node
  - 2) A way of creating an initial node from a problems 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 a corresponds to

- Different alternatives exist!

Total Order?

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

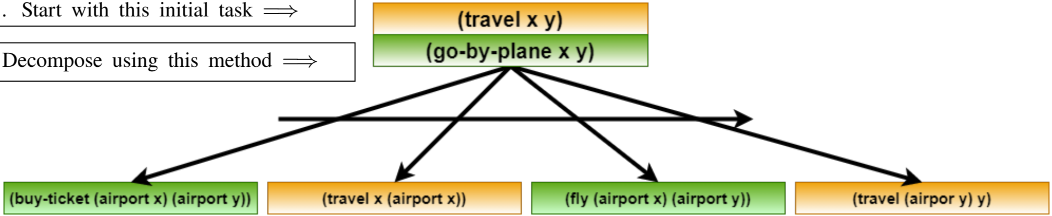


Forward Decomposition?

- Different decomposition orders are still possible

1. Start with this initial task $\Longrightarrow$

2. Decompose using this method $\Longrightarrow$

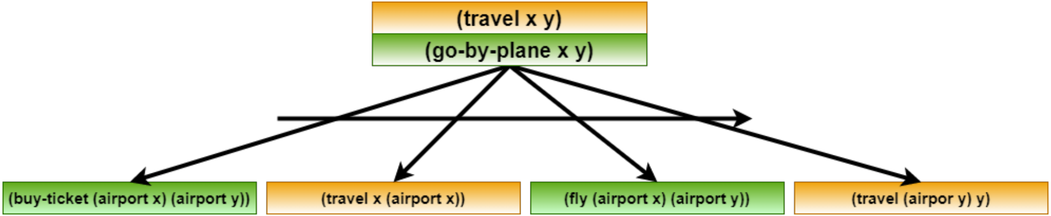


3. Multiple alternatives: which to decompose next?

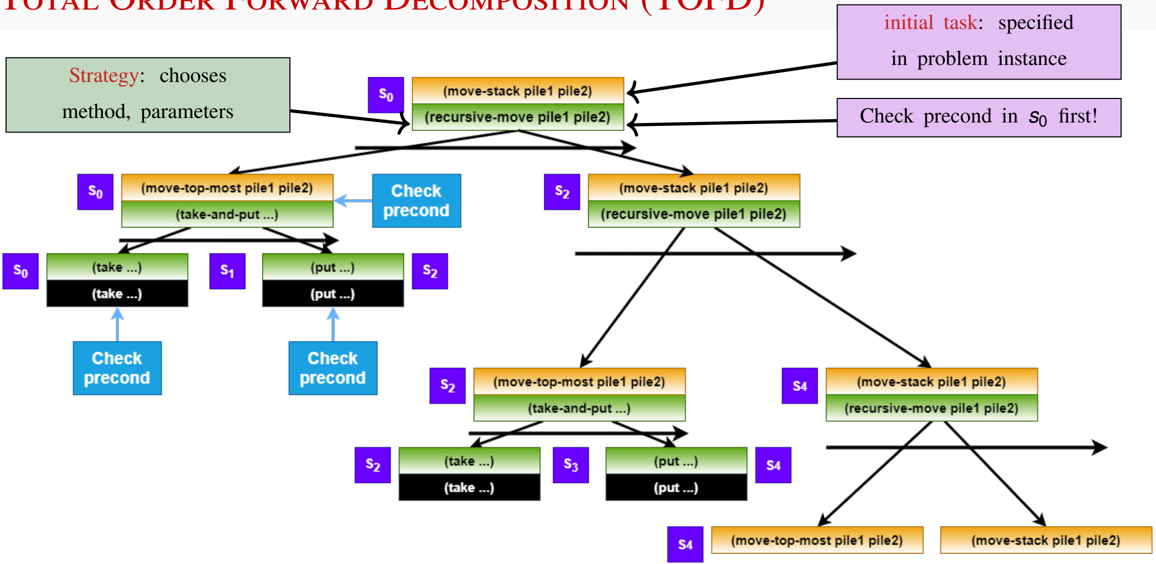
Choose what to decompose, which method to use, how to parameterize it $\Longrightarrow$ Need search!

Forward Decomposition!

- 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 will be executed
    - $\Longrightarrow$ When we decompose a task, we know the "current" state of the world!



Total Order Forward Decomposition (TOFD)

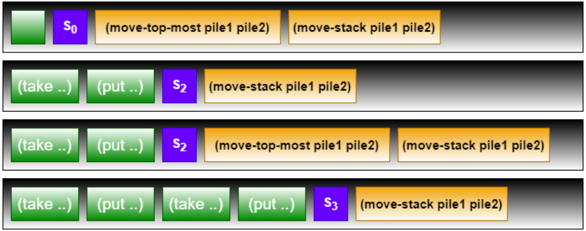


TOFD Node Structure

- [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 task from the problem!

- Examples: Nodes visited in the previous slide



TOFD Successors

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

- [4] Solution test
  - No more tasks $\Longrightarrow$ done!
- [5] Solution extraction
  - The resulting search node *contains* a sequential plan!

Solving Total Order STN Problems

- TOFD takes a search node
  - π - a sequence of actions
  - s - the current state
  - $\langle t_1, \ldots, t_k \rangle$ - a list of tasks to be achieved in the specific 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!

TOFD: Base case

function TOTAL-ORDER-FORWARD-DECOMPOSITION(problem)
  initial-node ← $\langle [], \mathit{problem.initialstate}, \mathit{problem.initialtask} \rangle$ → *[2]*
  open ← {initial-node}
  while (open $\neq \emptyset$) **do**
    $\langle \pi, s, \langle t_1, \ldots, t_k \rangle \rangle$ ← SEARCH-STRATEGY-REMOVE-FROM(open)

→ [6] TOFD uses depth first search
→ [4] If we have no tasks left to decompose..

    if $k = 0$ **then return** π

TOFD: GROUND PRIMITIVE TASKS

function TOTAL-ORDER-FORWARD-DECOMPOSITION(problem)

initial-node $\leftarrow \langle [], \mathit{problem.initialstate}, \mathit{problem.initialtask} \rangle$
open $\leftarrow$ {initial-node}
while (open $\neq \emptyset$) **do**
$\langle \pi, s, \langle t_1, \ldots, t_k \rangle \rangle \leftarrow$ SEARCH-STRATEGY-REMOVE-FROM(open)

if $k = 0$ **then return** π
if t_1 is primitive **then**
$\mathit{actions} \leftarrow$ GROUND-INSTANCE-OF-OPERATORS(O)
$\mathit{candidates} \leftarrow \{\, a \mid a \in \mathit{actions} \wedge \mathit{name}(a) = t_1 \wedge a$ applicable in $s\}$

$\rightarrow$ *For simplicity: The case when all tasks to achieve are ground*
$\rightarrow$ *[2]*
$\rightarrow$ *[6] TOFD uses depth first search*
$\rightarrow$ *[4] If we have no tasks left to decompose..*
$\rightarrow$ *A primitive task is decomposed into a single action! Possibly many to choose from!*

TOFD: Successors

function TOTAL-ORDER-FORWARD-DECOMPOSITION(problem) → *For simplicity: The case when all tasks to achieve are ground*

  initial-node $\leftarrow \langle [], \mathit{problem.initialstate}, \mathit{problem.initialtask} \rangle$ → *[2]*

  open $\leftarrow$ {initial-node}

  while (open $\neq \emptyset$) **do**

    $\langle \pi, s, \langle t_1, \ldots, t_k \rangle \rangle \leftarrow$ SEARCH-STRATEGY-REMOVE-FROM(open)

    if $k = 0$ **then return** π

    if t_1 is primitive **then** → *A primitive task is decomposed into a single action! Possibly many to choose from!*

      $\mathit{actions} \leftarrow$ GROUND-INSTANCE-OF-OPERATORS(O)

      $\mathit{candidates} \leftarrow \{ a | a \in \mathit{actions} \wedge \mathit{name}(a) = t_1 \wedge a$ applicable in $s \}$

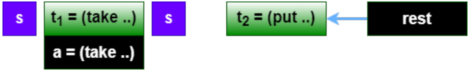
      for each $a \in \mathit{candidates}$ **do**

        $\pi' \leftarrow \pi + a$ → *Add action at the end*

        $s' \leftarrow \gamma(s, a)$ → *Apply the action, find the new state*

        $\mathit{rest} \leftarrow \langle t_2, \ldots, t_k \rangle$

        $\mathit{open} \leftarrow \mathit{open} \cup \{ \langle \pi', s', \mathit{rest} \rangle \}$



TOFD: LIFTED PRIMITIVE TASKS

function TOTAL-ORDER-FORWARD-DECOMPOSITION(problem)
  initial-node ← $\langle [], \mathit{problem.initialstate}, \mathit{problem.initialtask} \rangle$
  open ← {initial-node}
  while (open $\neq \emptyset$) **do**
    $\langle \pi, s, \langle t_1, \ldots, t_k \rangle \rangle$ ← SEARCH-STRATEGY-REMOVE-FROM(open)
    if $k = 0$ **then return** π
    if t_1 is primitive **then**
      $\mathit{actions} \leftarrow$ GROUND-INSTANCE-OF-OPERATORS(O)
      $\mathit{candidates} \leftarrow \{ (a, \sigma) | a \in \mathit{actions} \wedge \mathit{name}(a) = \sigma(t_1) \wedge a$ applicable in $s\}$

→ *The case when all tasks to achieve are non-ground. The plan will still be ground*

→ *[2]*

→ *A primitive task is decomposed into a single action! Possibly many to choose from!*

→ *σ is a substitution function! Basically, σ can specify variable bindings for parameters of t_1...*

S

t_1 = (take ?crane loc1 cont2 cont pile8)

t_2 = (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 }

TOFD: Lifted Primitive Tasks

function TOTAL-ORDER-FORWARD-DECOMPOSITION(problem)
    initial-node $\leftarrow \langle [], \mathit{problem.initialstate}, \mathit{problem.initialtask} \rangle$
    open $\leftarrow$ {initial-node}
    while (open $\neq \emptyset$) **do**
        $\langle \pi, s, \langle t_1, \ldots, t_k \rangle \rangle \leftarrow$ SEARCH-STRATEGY-REMOVE-FROM(open)
        if $k = 0$ **then return** π
        if t_1 is primitive **then**
            $\mathit{actions} \leftarrow$ GROUND-INSTANCE-OF-OPERATORS(O)
            $\mathit{candidates} \leftarrow \{ (a, \sigma) | a \in \mathit{actions} \wedge \mathit{name}(a) = \sigma(t_1) \wedge a$ applicable in $s\}$
            for each $a, \sigma \in \mathit{candidates}$ **do**
                $\pi' \leftarrow \pi + a$
                $s' \leftarrow \gamma(s, a)$
                $\mathit{rest} \leftarrow \langle \sigma(t_2), \ldots, \sigma(t_k) \rangle \rangle$
                $\mathit{open} \leftarrow \mathit{open} \cup \{ \langle \pi', s', \mathit{rest} \rangle \}$

$\rightarrow$ *The case when all tasks to achieve are non-ground. The plan will still be ground*

$\rightarrow$ *[2]*

$\rightarrow$ *A primitive task is decomposed into a single action! Possibly many to choose from!*

$\rightarrow$ *Add action at the end*

$\rightarrow$ *Apply the action, find the new state*

$\rightarrow$ *Must have the same variable bindings!*

s

t_1 = (take ?crane loc1 cont2 cont pile8)

$\sigma(t_2)$ = (put crane1)

chosen:

(take crane1 loc1 cont2 cont5 pile8) σ = { ?crane $\rightarrow$ crane1, ?cont $\rightarrow$ cont5 }

(take crane2 loc1 cont2 cont5 pile8) σ = { ?crane $\rightarrow$ crane2, ?cont $\rightarrow$ cont5 }

TOFD: Non-Primitive Tasks

function TOTAL-ORDER-FORWARD-DECOMPOSITION(problem)

→ *The case when all tasks to achieve are non-ground. The plan will still be ground*

initial-node ← $\langle [], \mathit{problem.initialstate}, \mathit{problem.initialtask} \rangle$ → *[2]*

open ← {initial-node}

while (open $\neq \emptyset$) **do**

$\langle \pi, s, \langle t_1, \ldots, t_k \rangle \rangle$ ← SEARCH-STRATEGY-REMOVE-FROM(open)

if $k = 0$ **then return** π

if t_1 is primitive **then**

...

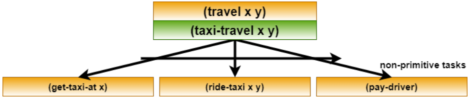
else

→ *t_1 is e.g.* `(travel LiU Resecentrum)`

→ *A non-primitive task is decomposed into a new task-list. May have many methods to choose from!*

ground ← GROUND-INSTANCES-OF-METHODS(M)

$\mathit{candidates}$ ← $\{ (m, \sigma) | m$ ground $\wedge\ \mathit{task}(m) = \sigma(t_1) \wedge m$ applicable in $s\}$



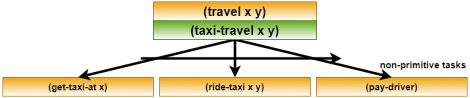
TOFD: Non-Primitive Tasks (cont.)

function TOTAL-ORDER-FORWARD-DECOMPOSITION(problem)

initial-node ← ⟨[], *problem.initialstate*, *problem.initialtask*⟩
open ← {initial-node}
while (open $\neq \emptyset$) **do**
  $\langle \pi, s, \langle t_1, \ldots, t_k \rangle \rangle \leftarrow$ SEARCH-STRATEGY-REMOVE-FROM(open)
  if $k = 0$ **then return** π
  if t_1 is primitive **then**
    ...
  else
    $ground \leftarrow$ GROUND-INSTANCES-OF-METHODS(M)
    $candidates \leftarrow \{ (m, \sigma) | m$ ground $\wedge\ task(m) = \sigma(t_1) \wedge m$ applicable in $s\}$
    for each $(m, \sigma) \in candidates$ **do**
      $\pi' \leftarrow \pi$
      $s' \leftarrow s$
      $rest \leftarrow \langle \text{SUBTASKS}(m) + \sigma(t_2), \ldots, \sigma(t_k) \rangle$
      $open \leftarrow open \cup \{\langle \pi', s', rest \rangle\}$

→ *The case when all tasks to achieve are non-ground. The plan will still be ground*
→ *[2]*

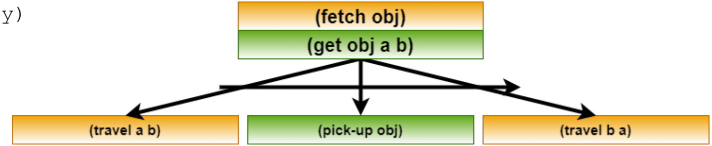
→ *No action needed!*
→ *No state change*
→ *Prepend new list! The "origin" of a task is discarded: only the sub-tasks are relevant!*



Limitations of Ordered-Task Planning

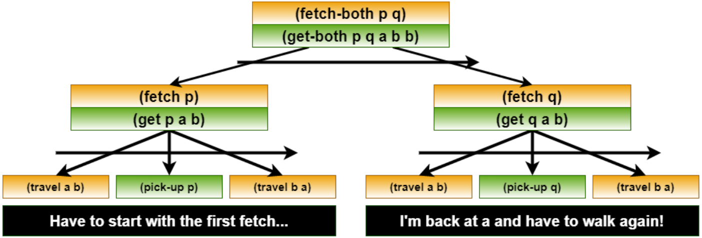
- TOFD requires totally ordered methods
  - Can't interleaves sub-tasks of different tasks
- Suppose we want to fetch one object somewhere, then return to where we are now
  - **task**: `(fetch obj)`
  - **method**: `(get obj mypos objpos)`
  - **precond**: `(robotat mypos) (at obj objpos)`
  - **subtasks**: ⟨ `(travel mypos objpos)`, `(pick-up obj)`, `(travel objpos mypos)` ⟩

  - **task**: `(travel x y)`
  - **method**: `(walk x y)`
  - **method**: `(stayat x y)`



Limitations of Ordered-Task Planning (cont.)

- Suppose we want to fetch two objects somewhere, then return to where we are now
- 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)` ⟩



Alternative methods

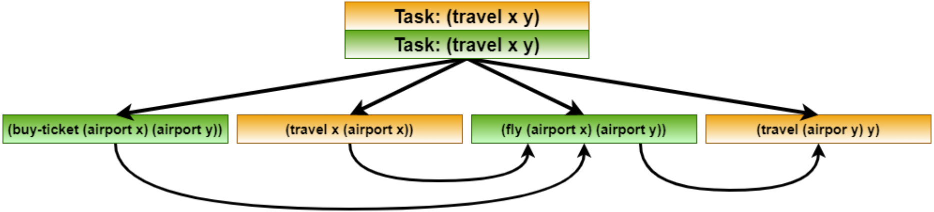
- To generate more efficient plans using total-order STNs:
  - Use a different domain model!
    - **task**: `(fetch-both obj1 obj2)`
    - **method**: `(get-both obj1 obj2 mypos objpos1 objpos2)`
    - **precond**: `(!= objpos1 objpos2) (at obj1 objpos1) (at obj2 objpos2)`
    - **subtasks**: ⟨ `(travel mypos objpos1)`, `(pick-up obj1)`, `(travel objpos1 objpos2)`, `(pick-up obj2)`, `(travel objpos2 mypos)` ⟩

    - **task**: `(fetch-both obj1 obj2)`
    - **method**: `(get-both-in-same-place obj1 obj2 mypos objpos)`
    - **precond**: `(at obj1 objpos) (at obj2 objpos)`
    - **subtasks**: ⟨ `(travel mypos objpos)`, `(pick-up obj1)`, `(pick-up obj2)`, `(travel objpos mypos)` ⟩

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

Partially Ordered Methods

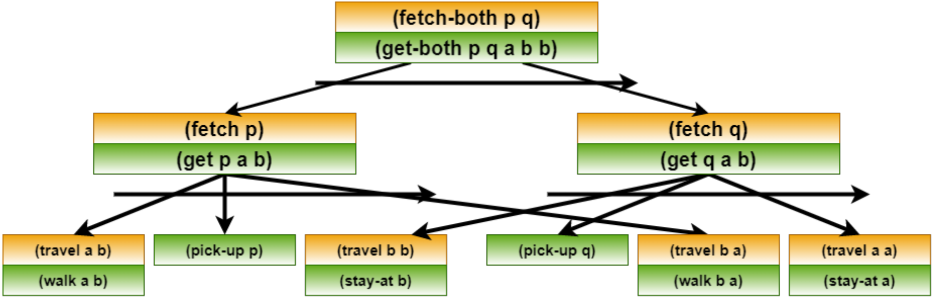
- The sub-tasks are a partially ordered set $\{t_1, \ldots t_k\}$ – a *network*



- **method**: `(go-by-plane x y)`
  - **task**: `(travel x y)`
  - **precond**: `(long-distance x y)`
  - **network**: $u_1 =$ `(buy-ticket (airport x) (airport y))`
    $u_2 =$ `(travel x (airport x))`
    $u_3 =$ `(fly (airport x) (airport y))`
    $u_4 =$ `(travel (airport y) y)`
    `{` $(u_1, u_3), (u_2, u_3), (u_3, u_4)$ `}`

Partially Ordered Methods

- With partially ordered methods sub-tasks can be interleaved



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

References I

[1] Hector Geffner and Blai Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2013. ISBN 9781608459698. doi: 10.2200/S00513ED1V01Y201306AIM022. URL `https://doi.org/10.2200/S00513ED1V01Y201306AIM022`.

[2] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning - theory and practice*. Elsevier, 2004. ISBN 978-1-55860-856-6.

[3] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016. ISBN 978-1-107-03727-4. URL `http://www.cambridge.org/de/academic/subjects/computer-science/artificial-intelligence-and-natural-language-processing/automated-planning-and-acting?format=HB`.