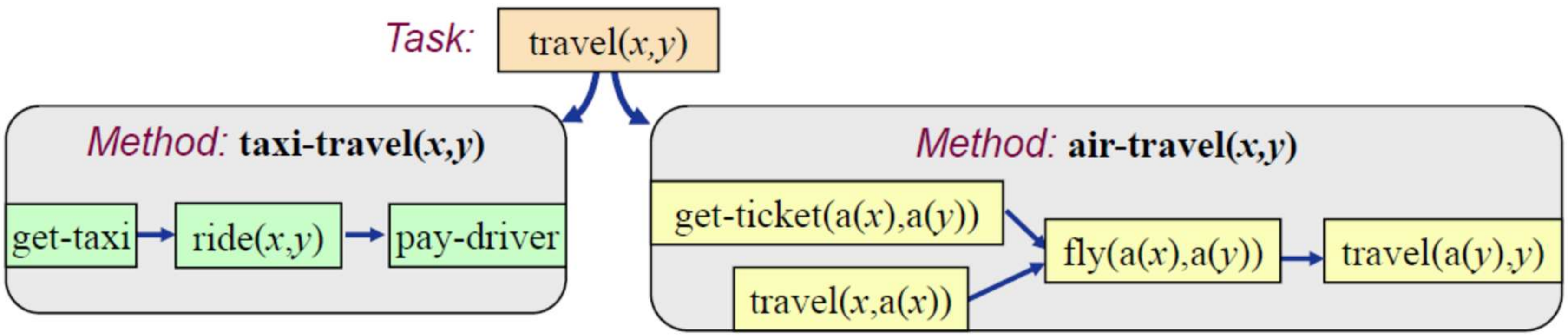
HTN Planning

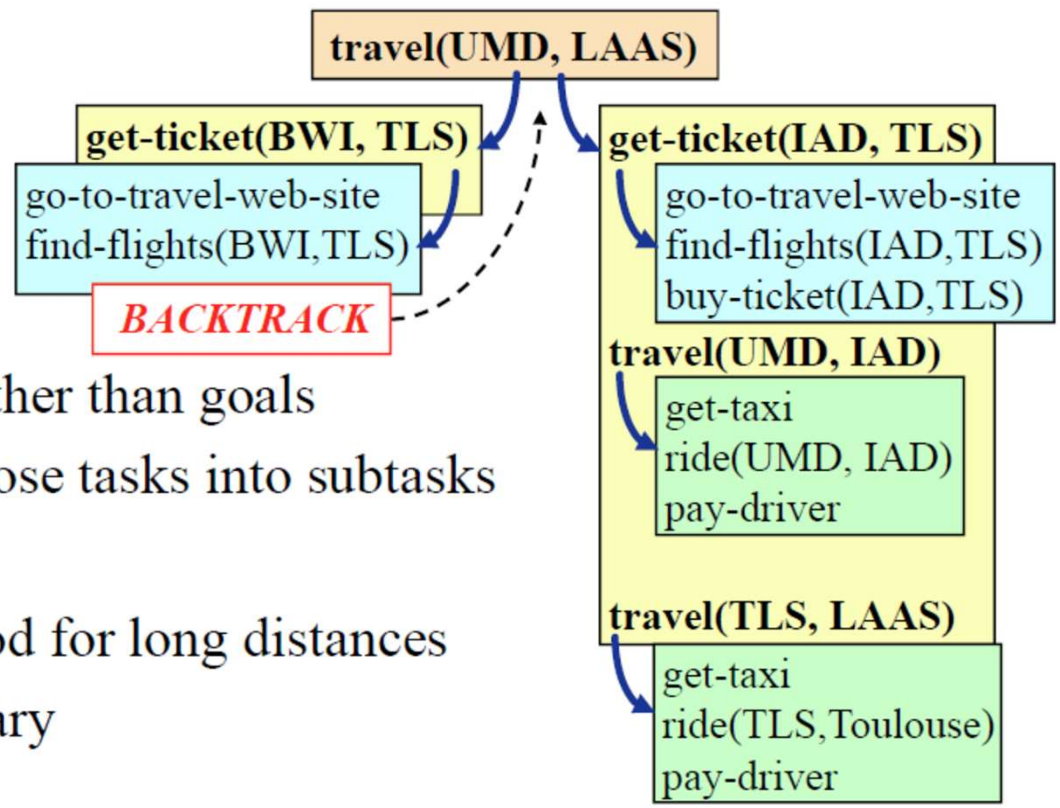
- Problem reduction
  - *Tasks* (activities) rather than goals
  - *Methods* to decompose tasks into subtasks
  - Enforce constraints
    - » E.g., taxi not good for long distances
  - Backtrack if necessary

HTN Planning

- More flexibility in modeling:
  - incorporate procedural expert knowledge (modeling means, speed up search)
- More complex behavior
  - pose complex restrictions on the desired solutions
- Easier user integration in the plan generation process
  - mixed initiative planning; MIP
- Communicate plans on different levels of abstraction
- Incorporate task abstraction in plan explanations



HTN Planning



- Problem reduction
  - *Tasks* (activities) rather than goals
  - *Methods* to decompose tasks into subtasks
  - Enforce constraints
    - » E.g., taxi not good for long distances
  - Backtrack if necessary

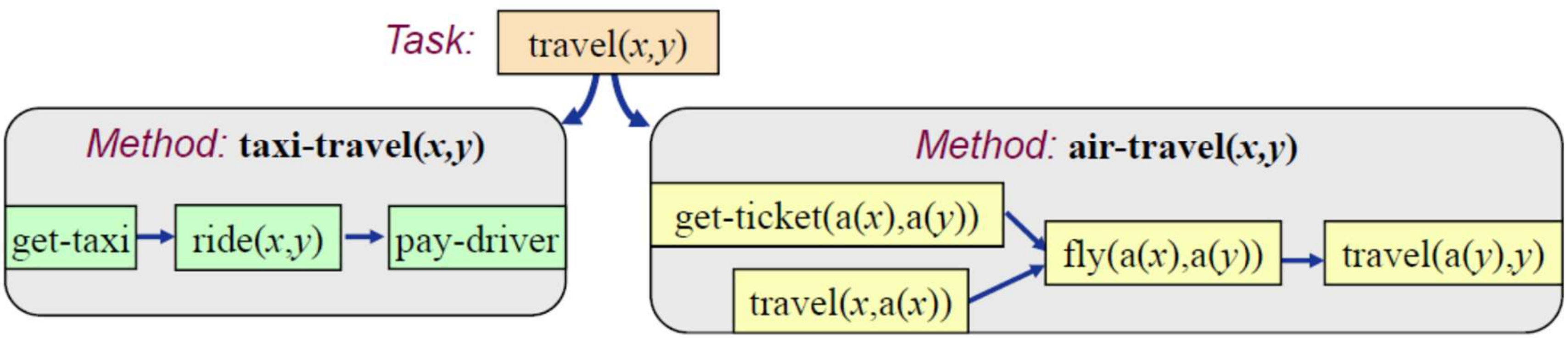
HTN Planning

- “HTN planners differ from classical planners in what they plan for and how they plan for it. In an HTN planner, the objective is not to achieve a set of goals but instead to perform some set of tasks.” (Ghallab, Nau, and Traverso; Automated Planning: Theory and Practice)

- Main differences to classical planning problems:
  - The goal is to find a refinement of the initial task(s), not to satisfy some goal description
  - No arbitrary task insertion: decompose compound tasks using their pre-defined methods

HTN Planning

- HTN planners may be domain-specific
- Or they may be domain-configurable
  - Domain-independent planning engine
  - Domain description that defines not only the operators, but also the methods
  - Problem description
    - » domain description, initial state, initial task network



Simple Task Network (STN) Planning

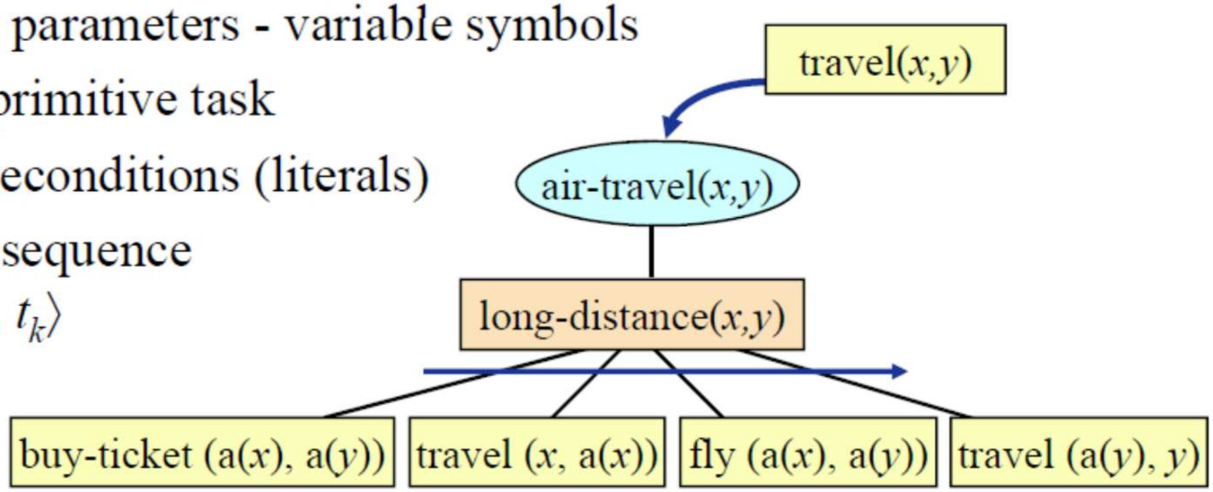
- A special case of HTN planning
- States and operators
  - The same as in classical planning
- *Task*: an expression of the form $t(u_1,\ldots,u_n)$
  - t is a *task symbol*, and each u_i is a term
  - Two kinds of task symbols (and tasks):
    - » *primitive*: tasks that we know how to execute directly
      - task symbol is an operator name
    - » *nonprimitive*: tasks that must be decomposed into subtasks
      - use *methods* (next slide)

Methods

- Totally ordered method: a 4-tuple

  $$m = (\text{name}(m), \text{task}(m), \text{precond}(m), \text{subtasks}(m))$$

  - name(m): an expression of the form $n(x_1,\ldots,x_n)$
    - $x_1,\ldots,x_n$ are parameters - variable symbols
  - task(m): a nonprimitive task
  - precond(m): preconditions (literals)
  - subtasks(m): a sequence of tasks $\langle t_1, \ldots, t_k \rangle$



air-travel(x,y)

task: travel(x,y)

precond: long-distance(x,y)

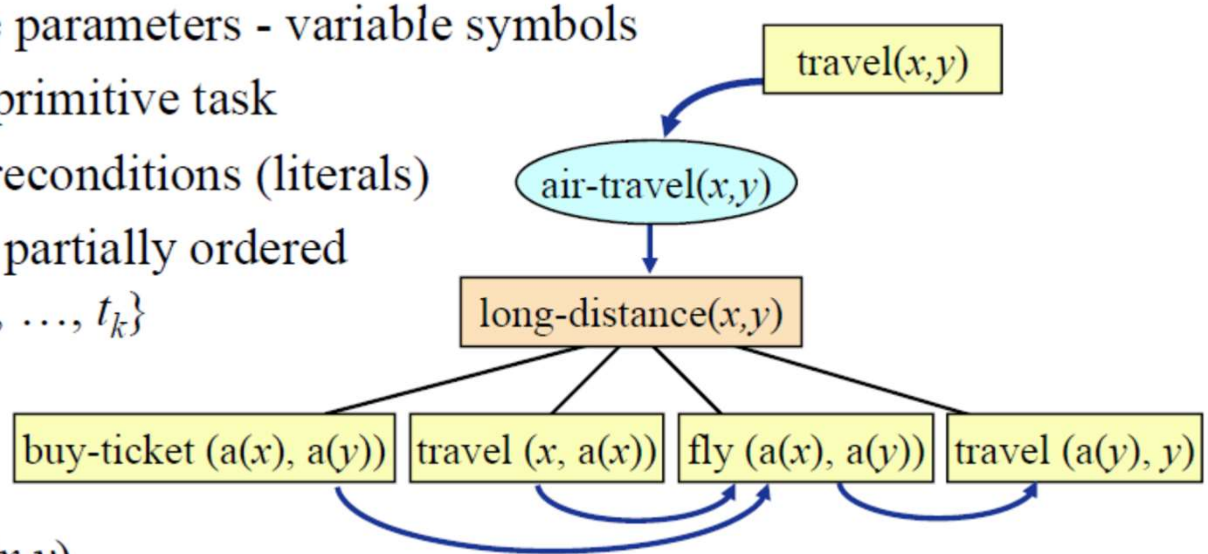
subtasks: $\langle$buy-ticket($a(x)$, $a(y)$), travel(x,$a(x)$), fly($a(x)$, $a(y)$), travel($a(y)$,y)$\rangle$

Methods (Continued)

- Partially ordered method: a 4-tuple

  m = (name(m), task(m), precond(m), subtasks(m))

  - name(m): an expression of the form $n(x_1,\ldots,x_n)$
    - » $x_1,\ldots,x_n$ are parameters - variable symbols
  - task(m): a nonprimitive task
  - precond(m): preconditions (literals)
  - subtasks(m): a partially ordered set of tasks $\{t_1, \ldots, t_k\}$



air-travel(x,y)

task: travel(x,y)

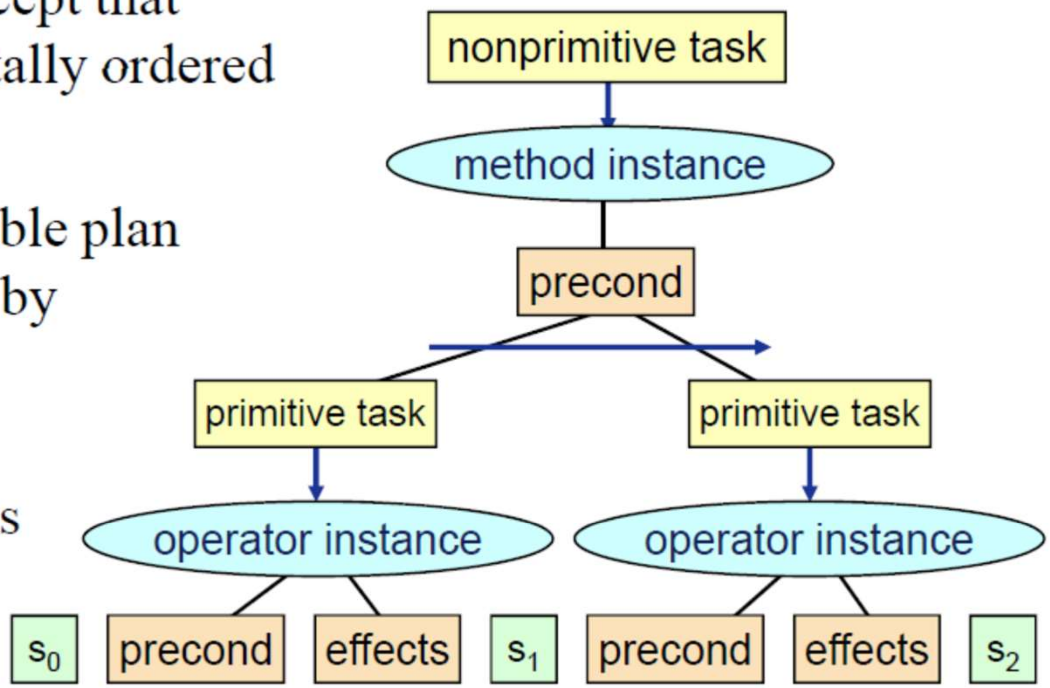
precond: long-distance(x,y)

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

u_4= travel($a(y),y$), $\{(u_1,u_3), (u_2,u_3), (u_3,u_4)\}$

Domains, Problems, Solutions

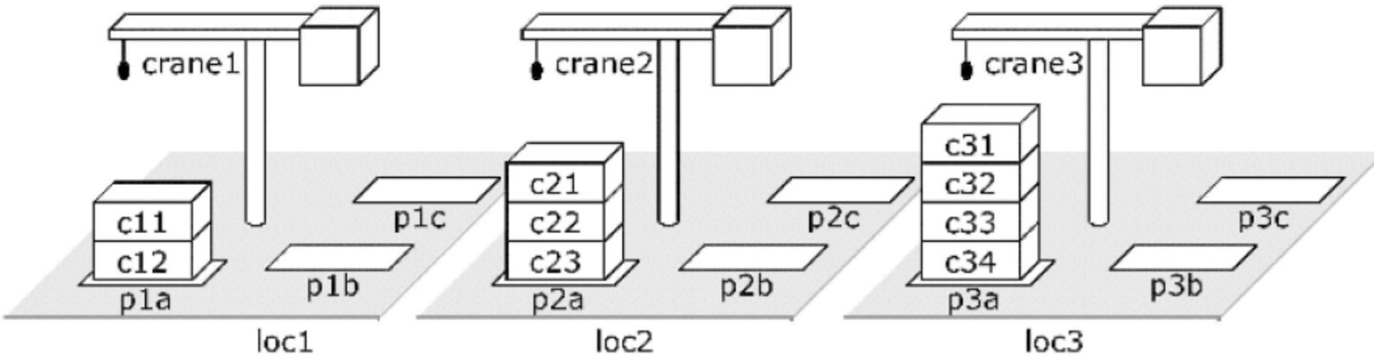
- STN planning domain: methods, operators
- STN planning problem: methods, operators, initial state, task list
- Total-order STN planning domain and planning problem:
  - Same as above except that all methods are totally ordered



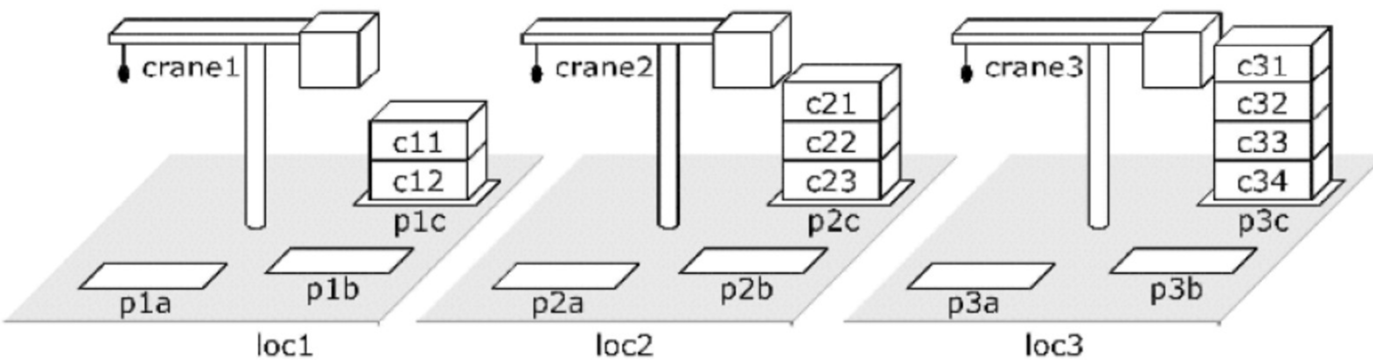
- Solution: any executable plan that can be generated by recursively applying
  - methods to non-primitive tasks
  - operators to primitive tasks

Example

- Suppose we want to move three stacks of containers in a way that preserves the order of the containers



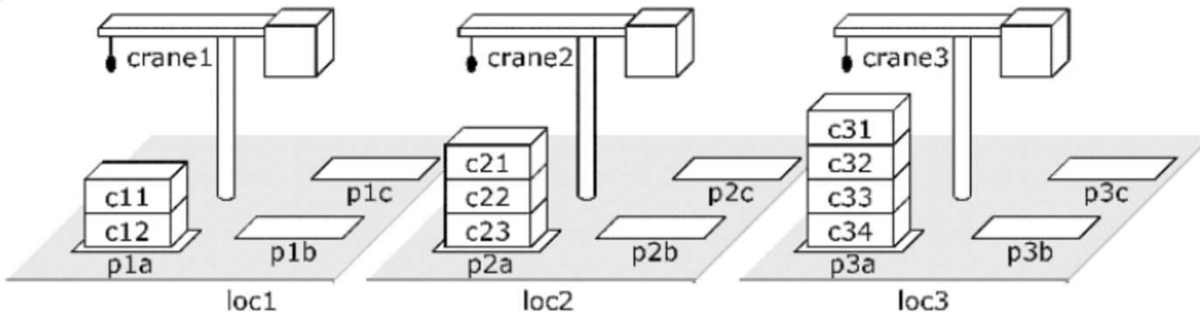
(a) initial state



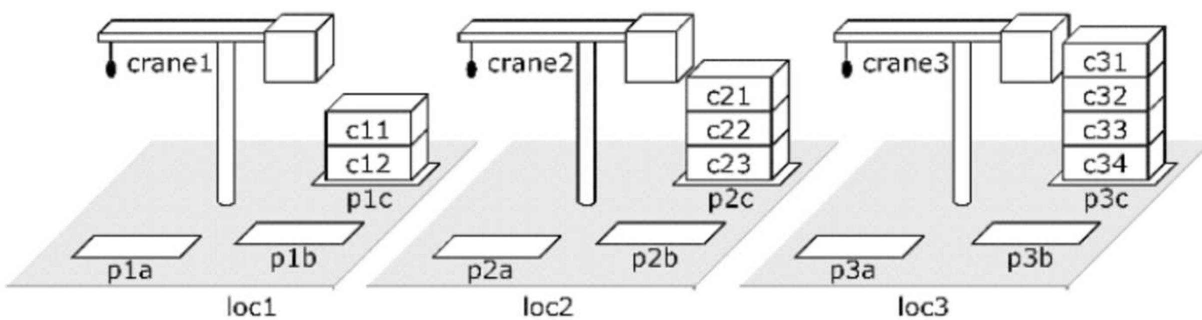
(b) goal

Example (continued)

- A way to move each stack:
  - first move the containers from p to an intermediate pile r
  - then move them from r to q



(a) initial state



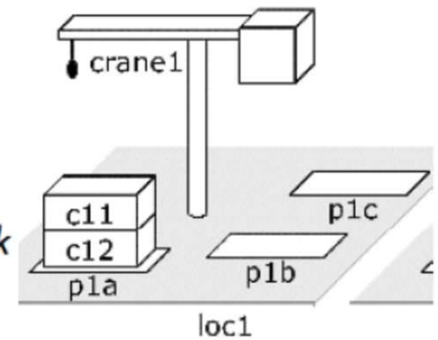
(b) goal

using crane k at location l, take container c from object x1 (container or pallet) in pile p1 and put it onto object x2 in pile p2

Total-Order Formulation

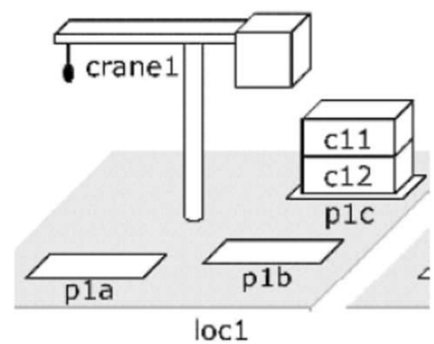
take-and-put$(c, k, l_1, l_2, p_1, p_2, x_1, x_2)$:
task: move-topmost-container(p_1, p_2)
precond: top(c, p_1), on(c, x_1), *; true if p_1 is not empty*
attached(p_1, l_1), belong(k, l_1), *; bind l_1 and k*
attached(p_2, l_2), top(x_2, p_2) *; bind l_2 and x_2*
subtasks: ⟨take(k, l_1, c, x_1, p_1), put(k, l_2, c, x_2, p_2)⟩

recursive-move(p, q, c, x):
task: move-stack(p, q)
precond: top(c, p), on(c, x) *; true if p is not empty*
subtasks: ⟨move-topmost-container(p, q), move-stack(p, q)⟩
;; the second subtask recursively moves the rest of the stack



do-nothing(p, q)
task: move-stack(p, q)
precond: top$(pallet, p)$ *; true if p is empty*
subtasks: ⟨⟩ *; no subtasks, because we are done*

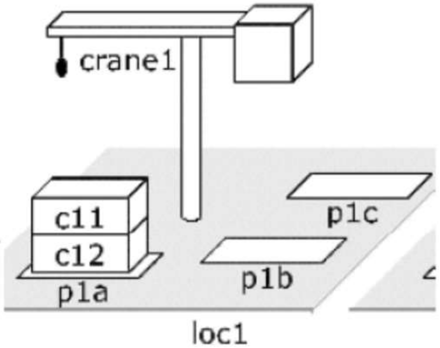
move-each-twice()
task: move-all-stacks()
precond: *; no preconditions*
subtasks: *; move each stack twice:*
⟨move-stack(p1a,p1b), move-stack(p1b,p1c),
move-stack(p2a,p2b), move-stack(p2b,p2c),
move-stack(p3a,p3b), move-stack(p3b,p3c)⟩



Partial-Order Formulation

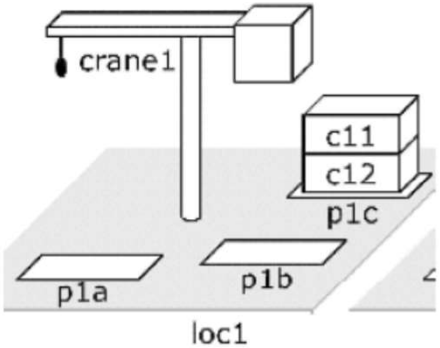
take-and-put$(c, k, l_1, l_2, p_1, p_2, x_1, x_2)$:
  task: move-topmost-container(p_1, p_2)
  precond: top(c, p_1), on(c, x_1), *; true if p_1 is not empty*
  attached(p_1, l_1), belong(k, l_1), *; bind l_1 and k*
  attached(p_2, l_2), top(x_2, p_2) *; bind l_2 and x_2*
  subtasks: $\langle$take(k, l_1, c, x_1, p_1), put$(k, l_2, c, x_2, p_2)\rangle$

recursive-move(p, q, c, x):
  task: move-stack(p, q)
  precond: top(c, p), on(c, x) *; true if p is not empty*
  subtasks: $\langle$move-topmost-container(p, q), move-stack$(p, q)\rangle$
  ;; the second subtask recursively moves the rest of the stack



do-nothing(p, q)
  task: move-stack(p, q)
  precond: top$(pallet, p)$ *; true if p is empty*
  subtasks: $\langle\rangle$ *; no subtasks, because we are done*

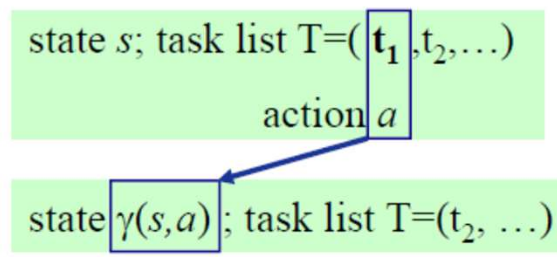
move-each-twice()
  task: move-all-stacks()
  precond: *; no preconditions*
  network: *; move each stack twice:*
  u_1 =move-stack(p1a,p1b), u_2 =move-stack(p1b,p1c),
  u_3 =move-stack(p2a,p2b), u_4 =move-stack(p2b,p2c),
  u_5 =move-stack(p3a,p3b), u_6 =move-stack(p3b,p3c),
  $\{(u_1, u_2), (u_3, u_4),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

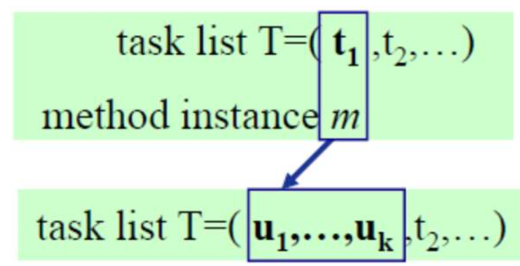


Solving Total-Order STN Planning Problems

Total-order Forward Decomposition (TFD)

TFD$(s, \langle t_1, \ldots, t_k \rangle, O, M)$
- if $k = 0$ then return $\langle\rangle$ (i.e., the empty plan)
- if t_1 is primitive then
  - $active \leftarrow \{(a, \sigma) \mid a$ is a ground instance of an operator in O,
    σ is a substitution such that a is relevant for $\sigma(t_1)$,
    and a is applicable to $s\}$
  - if $active = \emptyset$ then return failure
  - nondeterministically choose any $(a, \sigma) \in active$
  - $\pi \leftarrow$ TFD$(\gamma(s, a), \sigma(\langle t_2, \ldots, t_k \rangle), O, M)$
  - if π = failure then return failure
  - else return $a.\pi$
- else if t_1 is nonprimitive then
  - $active \leftarrow \{m \mid m$ is a ground instance of a method in M,
    σ is a substitution such that m is relevant for $\sigma(t_1)$,
    and m is applicable to $s\}$
  - if $active = \emptyset$ then return failure
  - nondeterministically choose any $(m, \sigma) \in active$
  - $w \leftarrow$ subtasks$(m).\sigma(\langle t_2, \ldots, t_k \rangle)$
  - return TFD(s, w, O, M)



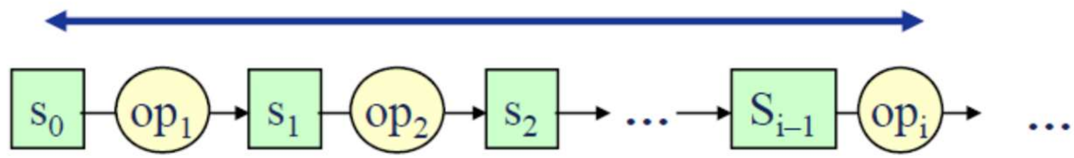


Applicability and Relevance

- A method instance m is <u>applicable</u> in a state s if
  - precond(m) satisfied in s
- A method instance m is <u>relevant</u> for a task t if
  - there is a substitution σ such that $\sigma(t) = task(m)$.
- The decomposition of a task t by a relevant method m under σ is
  - $\delta(t,m,\sigma) = \sigma(network(m))$ or
  - $\delta(t,m,\sigma) = \sigma(\langle subtasks(m) \rangle)$ if m is totally ordered.

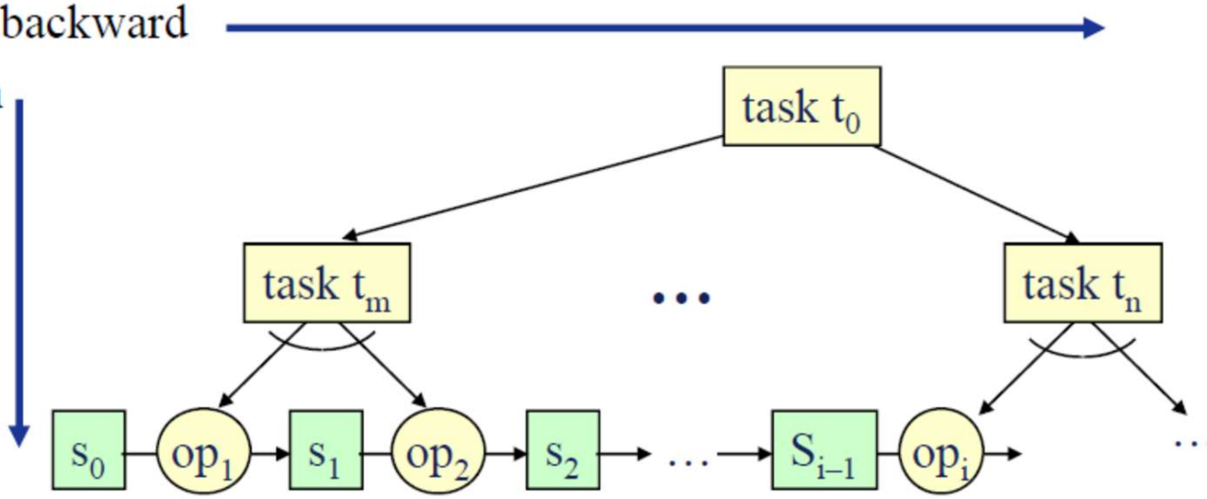
Comparison to Forward and Backward Search

- In state-space planning, must choose whether to search forward or backward



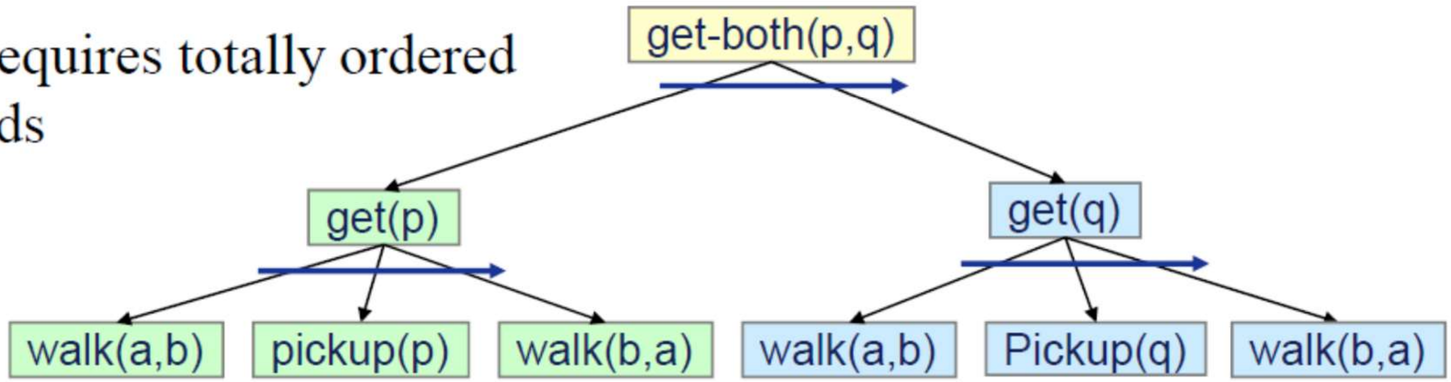
- In HTN planning, there are *two* choices to make about direction:
  - forward or backward
  - up or down

- TFD goes *down* and *forward*

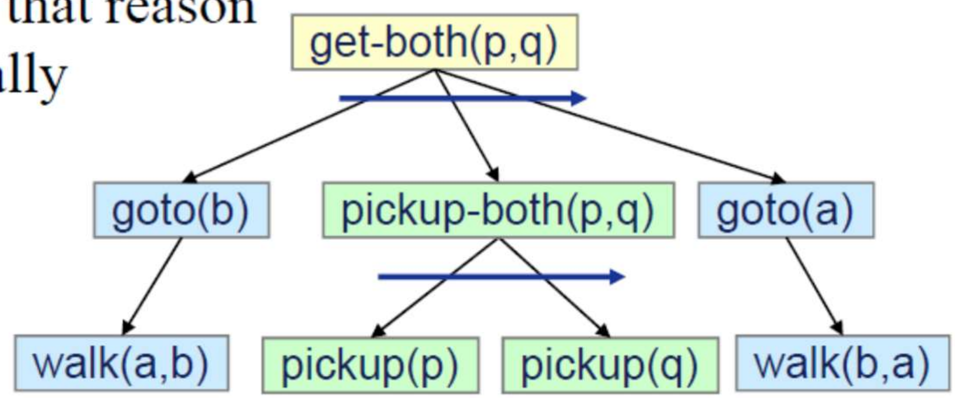


Limitation of Ordered-Task Planning

- TFD requires totally ordered methods

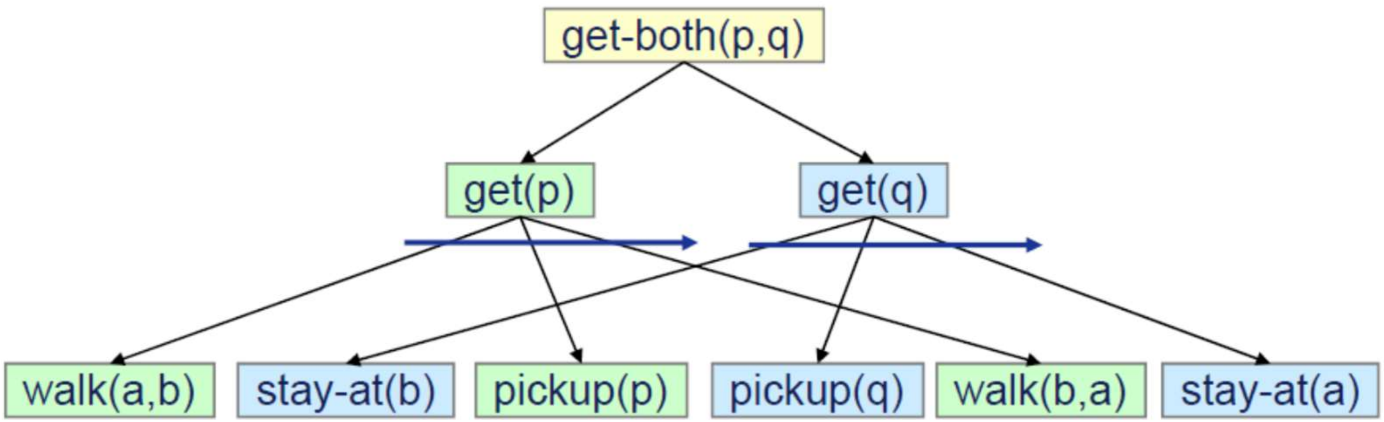


- Can't interleave subtasks of different tasks
- Sometimes this makes things awkward
  - Need to write methods that reason globally instead of locally



Partially Ordered Methods

- With partially ordered methods, the subtasks can be interleaved



- Fits many planning domains better
- Requires a more complicated planning algorithm

Algorithm for Partial-Order STNs

Partial-order Forward Decomposition (TFD)

PFD(s, w, O, M)
  if $w = \emptyset$ then return the empty plan
  nondeterministically choose any $u \in w$ that has no predecessors in w
  if t_u is a primitive task then
    active $\leftarrow \{(a, \sigma) \mid a$ is a ground instance of an operator in O,
      σ is a substitution such that $\text{name}(a) = \sigma(t_u)$,
      and a is applicable to $s\}$
    if *active* $= \emptyset$ then return failure
    nondeterministically choose any $(a, \sigma) \in$ *active*
    $\pi \leftarrow$ PFD$(\gamma(s, a), \sigma(w - \{u\}), O, M)$
    if $\pi =$ failure then return failure
    else return $a.\pi$
  else
    active $\leftarrow \{(m, \sigma) \mid m$ is a ground instance of a method in M,
      σ is a substitution such that $\text{name}(m) = \sigma(t_u)$,
      and m is applicable to $s\}$
    if *active* $= \emptyset$ then return failure
    nondeterministically choose any $(m, \sigma) \in$ *active*
    nondeterministically choose any task network $w' \in \delta(w, u, m, \sigma)$
    return(PFD(s, w', O, M))

$\pi = \{a_1, \ldots, a_k\}$; $w = \{\boxed{\mathbf{t_1}}, t_2, t_3 \ldots\}$
operator instance $\boxed{\boldsymbol{a}}$

$\pi = \{a_1 \ldots, a_k, \boxed{\boldsymbol{a}}\}$; $w' = \{t_2, t_3, \ldots\}$

$w = \{\boxed{\mathbf{t_1}}, t_2, \ldots\}$
method instance $\boxed{\boldsymbol{m}}$

$w' = \{\boxed{\mathbf{t_{11}, \ldots, t_{1k}}}, t_2, \ldots\}$

Algorithm for Partial-Order STNs

PFD(s, w, O, M)
    if $w = \emptyset$ then return the empty plan
    nondeterministically choose any $u \in w$ that has no predecessors in w

- Intuitively, w is a partially ordered set of tasks $\{t_1, t_2, \ldots\}$
    - But w may contain a task more than once
        - » e.g., travel from UMD to LAAS twice
    - The mathematical definition of a set doesn't allow this
- Define w as a partially ordered set of *task nodes* $\{u_1, u_2, \ldots\}$
    - Each task node u corresponds to a task t_u
- In my explanations, I talk about t and ignore u

),
(t_u),

$_k\}$; $w=\{\boxed{\mathbf{t_1}}, t_2, t_3 \ldots\}$
instance $\boxed{a}$

$\boxed{a}\}$; $w' = \{t_2, t_3, \ldots\}$

        else return $a.\pi$
    else
        active $\leftarrow \{(m, \sigma) \mid m$ is a ground instance of a method in M,
                    σ is a substitution such that name$(m) = \sigma(t_u)$,
                    and m is applicable to $s\}$
        if *active* $= \emptyset$ then return failure
        nondeterministically choose any $(m, \sigma) \in$ *active*
        nondeterministically choose any task network $w' \in \delta(w, u, m, \sigma)$
        return(PFD(s, w', O, M))

$w=\{\boxed{\mathbf{t_1}}, t_2, \ldots\}$
method instance $\boxed{m}$

$w' = \{\boxed{\mathbf{t_{11}, \ldots, t_{1k}}}, t_2, \ldots\}$

Algorithm for Partial-Order STNs

PFD(s, w, O, M)

if $w = \emptyset$ then return the empty plan

nondeterministically choose any $u \in w$ that has no predecessors in w

if t_u is a primitive task then

active $\leftarrow \{(a, \sigma) \mid a$ is a ground instance of an operator in O,
σ is a substitution such that $\text{name}(a) = \sigma(t_u)$,
and a is applicable to $s\}$

if *active* $= \emptyset$ then return failure

nondeterministically choose any $(a, \sigma) \in$ *active*

$\pi \leftarrow$ PFD$(\gamma(s, a), \sigma(w - \{u\}), O, M)$

if $\pi =$ failure then return failure

else return $a.\pi$

else

active $\leftarrow \{(m, \sigma) \mid m$ is a ground instance of a method in M,
σ is a substitution such that $\text{name}(m) = \sigma(t_u)$,
and m is applicable to $s\}$

if *active* $= \emptyset$ then return failure

nondeterministically choose any $(m, \sigma) \in$ *active*

nondeterministically choose any task network $w' \in \delta(w, u, m, \sigma)$

return(PFD(s, w', O, M))

$\pi = \{a_1, \ldots, a_k\}$; $w = \{\boxed{\mathbf{t_1}}, t_2, t_3 \ldots\}$

operator instance $\boxed{\boldsymbol{a}}$

$\pi = \{a_1 \ldots, a_k, \boxed{\boldsymbol{a}}\}$; $w' = \{t_2, t_3, \ldots\}$

$w = \{\boxed{\mathbf{t_1}}, t_2, \ldots\}$

method instance $\boxed{\boldsymbol{m}}$

$w' = \{\boxed{\mathbf{t_{11}, \ldots, t_{1k}}}, t_2, \ldots\}$

Algorithm for Partial-Order STNs

PFD(s, w, O, M)
  if $w = \emptyset$ then return the empty plan
  nondeterministically choose any $u \in w$ that has no predecessors in w
  if t_u is a primitive task then
    active ←
    if *active*
    nondeter
    $\pi \leftarrow$ PF
    if π = failure then return failure
    else return $a.\pi$
  else
    active ← $\{(m, \sigma) \mid m$ is a ground instance of a method in M,
      σ is a substitution such that name$(m) = \sigma(t_u)$,
      and m is applicable to $s\}$
    if *active* $= \emptyset$ then return failure
    nondeterministically choose any $(m, \sigma) \in$ *active*
    nondeterministically choose any task network $w' \in \delta(w, u, m, \sigma)$
    return(PFD(s, w', O, M))

δ(*w, u, m,* σ) has a complicated definition in the book. Here's what it means:

- We nondeterministically selected t_1 as the task to do first
- Must do t_1's first subtask before the first subtask of every $t_i \neq t_1$
- Insert ordering constraints to ensure that this happens

$\pi=\{a_1 \ldots, a_k, \boxed{a}\};\ w'=\{t_2,t_3 \ldots\}$

$w=\{\boxed{t_1},t_2,\ldots\}$

method instance $\boxed{m}$

$w'=\{\boxed{t_{11},\ldots,t_{1k}},t_2,\ldots\}$

Comparison to Classical Planning

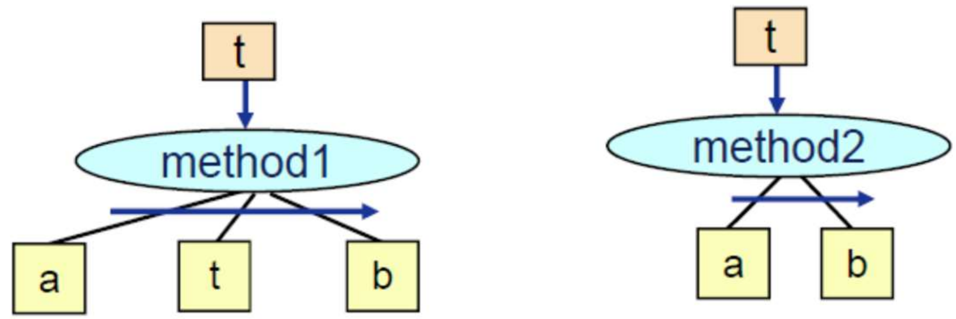
STN planning is strictly more expressive than classical planning

- Any classical planning problem can be translated into an ordered-task-planning problem in polynomial time
- Several ways to do this. One is roughly as follows:
  - For each goal or precondition e, create a task t_e
  - For each operator o and effect e, create a method $m_{o,e}$
    - » Task: t_e
    - » Subtasks: t_{c1}, t_{c2}, …, t_{cn}, o, where c_1, c_2, …, c_n are the preconditions of o
    - » Partial-ordering constraints: each t_{ci} precedes o

- (I left out some details, such as how to handle deleted-condition interactions)

Comparison to Classical Planning (cont.)

- Some STN planning problems aren't expressible in classical planning
- Example:
  - Two STN methods:
    - » No arguments
    - » No preconditions



  - Two operators, a and b
    - » Again, no arguments and no preconditions
  - Initial state is empty, initial task is t
  - Set of solutions is $\{a^n b^n \mid n > 0\}$
  - No classical planning problem has this set of solutions
    - » The state-transition system is a finite-state automaton
    - » No finite-state automaton can recognize $\{a^n b^n \mid n > 0\}$
- Can even express undecidable problems using STNs

SHOP2

- SHOP2: implementation of PFD-like algorithm + generalizations
  - Won one of the top four awards in the AIPS-2002 Planning Competition
  - Freeware, open source
  - Implementation available at

    `http://www.cs.umd.edu/projects/shop`

HTN Planning

- HTN planning is even more general
  - Can have constraints associated with tasks and methods
    - Things that must be true before, during, or afterwards
  - Some algorithms use causal links and threats like those in PSP

HTN Planning

- Hierarchical Task Networks generalise Simple Task Networks:
  - no forward decomposition is necessary, a task network w consists of a set of task nodes and a set of constraints

- (HTN Planning Problem) An HTN planning problem is a 3-tuple $P = (s_0, w_0, D)$ where s_0 is the initial state, w_0 is a task network called the initial task network, and D is the HTN planning domain which consists of a set of operators and methods.

- A plan $\pi = [a_1, \ldots, a_k]$ is a solution for a planning problem if there is a ground instance (U_0, C_0) of (U, C) and a total ordering $[u_1, \ldots, u_k]$ of the nodes of U0 such that
  - the plan π is executable in s_o and the total ordering fulfills all constraints.

HTN Planning

- Hierarchical Task Networks generalise Simple Task Networks:
  - no forward decomposition is necessary, a task network w consists of a set of task nodes and a set of constraints

- A (hierarchical) task network is a pair w=(U,C), where:
- U is a set of tasks and
- C is a set of constraints of the following types:
  - t1≺t2: precedence constraint between tasks satisfied if in every solution π: last({t},π) ≺ first({t},π);
  - before(U',l): satisfied if in every solution π: literal l holds in the state just before first(U',π);
  - after(U',l): satisfied if in every solution π: literal l holds in the state just after last(U',π);
  - between(U',U'',l): satisfied if in every solution π: literal l holds in every state after last(U',π) and before first(U'',π).

<u>first(U',π)</u> = the action $a_i \in A(U')$ that occurs first in π;
and
<u>last(U',π)</u> = the action $a_i \in A(U')$ that occurs last in π.

HTN Planning

- Hierarchical Task Networks generalise Simple Task Networks:
  - no forward decomposition is necessary, a task network w consists of a set of task nodes and a set of constraints

- Let MS be a set of method symbols. An HTN method is a 4-tuple m=(name(m),task(m),subtasks(m),constr(m)) where:
  - name(m): the name of the method
    - syntactic expression of the form n(x1,…,xk)
    - n∈MS: method symbol
    - x1,…,xk: variable symbols that occur in m;
  - task(m): a non-primitive task;
  - (subtasks(m),constr(m)): a task network.

HTN Planning

take-and-put(c,k,l,p_o,p_d,x_o,x_d)

- task: move-topmost(p_o,p_d)
- network:
  - subtasks: {t_1=take(k,l,c,x_o,p_o), t_2=put(k,l,c,x_d,p_d)}
  - constraints: {$t_1 \prec t_2$, before({t_1}, top(c,p_o)), before({t_1}, on(c,x_o)), before({t_1}, attached(p_o,l)), before({t_1}, belong(k,l)), before({t_2}, attached(p_d,l)), before({t_2}, top(x_d,p_d))}

recursive-move(p_o,p_d,c,x_o)

- task: move-stack(p_o,p_d)
- network:
  - subtasks: {t_1=move-topmost(p_o,p_d), t_2=move-stack(p_o,p_d)}
  - constraints: {$t_1 \prec t_2$, before({t_1}, top(c,p_o)), before({t_1}, on(c,x_o))}

move-one(p_o,p_d,c)

- task: move-stack(p_o,p_d)
- network:
  - subtasks: {t_1=move-topmost(p_o,p_d)}
  - constraints: {before({t_1}, top(c,p_o)), before({t_1}, on(c,pallet))}

HTN Planning

Let (U,C) be a primitive HTN. A plan $\pi = \langle a_1,\ldots,a_n\rangle$ is a solution for $\mathcal{P}=(s_i,(U,C),O,M)$ if there is a ground instance $(\sigma(U),\sigma(C))$ of (U,C) and a total ordering $\langle t_1,\ldots,t_n\rangle$ of tasks in $\sigma(U)$ such that:

- for $i=1\ldots n$: name$(a_i) = t_i$;
- π is executable in s_i, i.e. $\gamma(s_i,\pi)$ is defined;
- the ordering of $\langle t_1,\ldots,t_n\rangle$ respects the ordering constraints in $\sigma(C)$;
- for every constraint before(U',l) in $\sigma(C)$ where t_k=first(U',π): l must hold in $\gamma(s_i, \langle a_1,\ldots,a_{k-1}\rangle)$;
- for every constraint after(U',l) in $\sigma(C)$ where t_k=last(U',π): l must hold in $\gamma(s_i, \langle a_1,\ldots,a_k\rangle)$;
- for every constraint between(U',U'',l) in $\sigma(C)$ where t_k=first(U',π) and t_m=last(U'',π): l must hold in every state $\gamma(s_i, \langle a_1,\ldots,a_j\rangle)$, $j\in\{k\ldots m-1\}$.

HTN Planning

Let $w = (U,C)$ be a non-primitive HTN. A plan $\pi = \langle a_1,\ldots,a_n\rangle$ is a solution for $\mathcal{P}=(s_i,w,O,M)$ if there is a sequence of task decompositions that can be applied to w such that:

- the result of the decompositions is a primitive HTN w'; and
- π is a solution for $\mathcal{P}'=(s_i,w',O,M)$.

HTN Planning

function Abstract-HTN(s,U,C,O,M)
  if (U,C).isInconsistent() **then return** failure
  if U.isPrimitive() **then**
    return extractSolution(s,U,C,O)
  else
    return decomposeTask(s,U,C,O,M)

function extractSolution(s,U,C,O)
  $\langle t_1,\ldots,t_n \rangle \leftarrow U$.chooseSequence($C$)
  $\langle a_1,\ldots,a_n \rangle \leftarrow$
    $\langle t_1,\ldots,t_n \rangle$.chooseGrounding($s,C,O$)
  if $\langle a_1,\ldots,a_n \rangle$.satisfies($C$) **then**
    return $\langle a_1,\ldots,a_n \rangle$
  return failure

function decomposeTask(s,U,C,O,M)
  $t \leftarrow U$.nonPrimitives().selectOne()
  $methods \leftarrow \{(m,\sigma) \mid m \in M$ and $\sigma(\text{task}(m)) = \sigma(t)\}$
  if $methods$.isEmpty() **then return** failure
  $(m,\sigma) \leftarrow methods$.chooseOne()
  $(U',C') \leftarrow \delta((U,C),t,m,\sigma)$
  $(U',C') \leftarrow (U',C')$.applyCritic()
  return Abstract-HTN(s,U',C',O,M)

Domain-Configurable Planners Compared to Classical Planners

- Disadvantage: writing a knowledge base can be more complicated than just writing classical operators
- Advantage: can encode “recipes” as collections of methods and operators
  - Express things that can’t be expressed in classical planning
  - Specify standard ways of solving problems
    - » Otherwise, the planning system would have to derive these again and again from “first principles,” every time it solves a problem
    - » Can speed up planning by many orders of magnitude (e.g., polynomial time versus exponential time)

HTN Planning

- HTN are particularly well-suited for planning in dynamic worlds (e.g., robotics [Bevacqua et al. 2015], games [Neil Wallace 2004, p.235])
- Planning is performed at multiple levels within a hierarchy.
- The search space is reduced. Invalid plans can often be ruled out early on. Hierarchical planners support replanning on the fly and can be used in dynamic worlds.

HTN Extension

- Hybrid Planning
- Task Insertion
- Temporal/resource constraints
- State Abstraction
- …