Part IX

Hierarchical Task Network Planning

Introduction

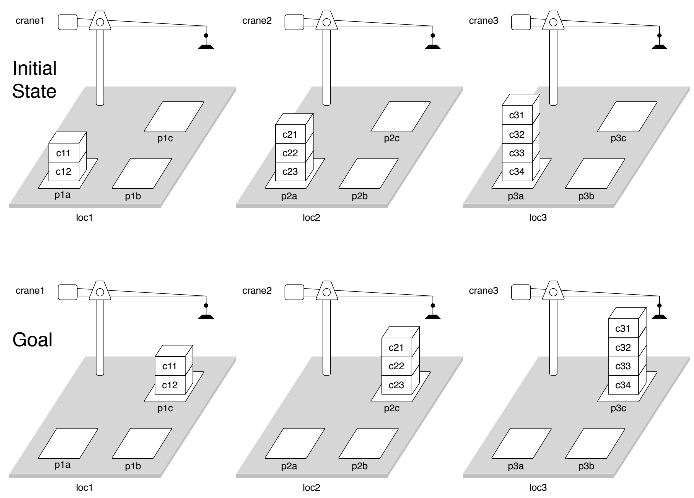
- Hierarchical Task Network (HTN) planning is like classical planning:
  - each state of the world is represented by a set of atoms
  - each action corresponds to a deterministic state transition
- In HTN planner, the objective is not to achieve a set of goals but instead to perform some set of tasks
- The imput to the HTN planning system includes
  1. a set of operators (similar to classical planning)
  2. a set of methods each of which is a presciption for how to decompose some task into some set of subtasks (smaller tasks)
- HTN planning has been more widely used for practical applications because HTN methods provide a convenient way to write problem-solving "recipes" that correspond to human expertise.

HTN Principle

HTN Principle

HTN planning proceeds by decomposing nonprimitive tasks recursively into smaller and smaller subtasks, until primitive tasks are reached that can be performed directly using the planning operators.

HTN Example (1/2)



HTN Example (2/2)

Example (Take and put method)

take-and-put(*c,k,l1,l2,p1,p2,x1,x2*)

precond: top(*p1,l1*), on(*c,x1*) *;; true if p1 is not empty*

attached(*p1,l1*), belong(*k,l1*) *;; bind l1 and k*

attached(*p2,l2*), top(*x2,p2*) *;; bind l2 and x2*

subtasks: ⟨take(*k,l1,c,x1,p1*), put(*k,l2,c,x2,p2*)⟩

To accomplish the task of moving the topmost container of a pile *p1* to another pile *p2*, we can use :

1. the DWR domain's take operator to remove the container from *p1* and
2. the put operator to put it on the top.

STN Planning

STN Planning

- STN (Simple Task Network) is a simplified version of HTN
- In STN, terms, literals, operators, actions and plans definitions are the same as in classical planning
- However, STN language includes:
  1. tasks
  2. methods
  3. task networks

Tasks Definition

frametitle

Definition (Task)

A task is an expression of the form

$$t(r_1, \ldots, r_k)$$

such

- t is a task symbol, i.e., an operator symbol (primitive task) or a method symbol (nonprimitive task)
- $r_1, \ldots, r_k$ are terms

Notes

1. A task is ground is all of the terms are ground; otherwise, it is unground
2. An action $a = (name(a), precond(a), effects(a))$ accomplishes a ground primitive task t in a state s if $name(a) = t$ and a is applicable to s.

Task Networks Definition

Definition (Simple Task Network)

A simple task network is an acyclic digraph

$$w = (U, E)$$

in which

- U is the node set such that each node $u \in U$ contains a task t_u
- E is the edge set that defines a partial ordering of U, e.g., $u \prec v$ iff there is a path from u to v

Notes

1. w is ground is all of the tasks $\{t_u \mid u \in U\}$ are ground; otherwise w is unground
2. w is primitive is all of the tasks $\{t_u \mid u \in U\}$ are primitive; otherwise w is nonprimitive

Task Networks Example

Example (Task Network)

In the DWR domain, let three tasks:

- t_1 = take(cran2,loc1,c1,c2,p1) a primitive task
- t_2 = put(cran2,loc2,c3,c4,p2) a primitive task
- t_3 = move-stack(p1,q) a non primitive task

and two task networks such $\forall i, u_i = ti$:

- $w_1 = (\{u1, u2, u3\}, \{(u1, u2), (u2, u3)\})$
- $w_2 = (\{u1, u2\}, \{(u1, u2)\})$

Since w_2 is totally ordered, we would usually write $w_2 = \langle t_1, t_2 \rangle$
Since w_2 is ground and primitive, it corresponds to the plan
⟨take(cran2,loc1,c1,c2,p1), put(cran2,loc2,c3,c4,p2)⟩

STN Method Definition

Definition

An STN method is a 4-tuple

$$m = (name(m), task(m), precond(m), network(m))$$

in which

- $name(m)$, the name of the method, i.e., a expression if the form $m(x_1, \ldots, x_2)$ where n is an unique method symbol and $x_1, \ldots, x_2$ are all of the variables symbols that occurs anywhere in m
- $task(m)$ is a non primitive task
- $precond(m)$ is a set of literals call method's preconditions
- $network(m)$ is a task network whose tasks are called the subtasks of m

STN Method Example (1/2)

Example (DWR methods)

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), on(c,x) *;; true if p is empty*
subtasks: ⟨⟩ *;; no substasks because we are done*

STN Method Example (2/2)

Example (DWR methods)

move-each-twice()

task: move-all-stacks()
precond: *;; no preconditions*
network: 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),
$\{(u_1, u_2), (u_3, u_4), (u_5, u_6)\}$

Applicale and Relevant Method

Definition (Applicable Method)

A method instance m is applicable in a state s if $\text{precond}^+(m) \subseteq s$ and $\text{precond}^-(m) \cap s = \emptyset$.

Definition (Revelant Method)

Let t be a task and m a method instance, if there is a substitution σ such that $\sigma(t) = task(m)$, then m is revelant for t, and the decomposition of t by m under σ is $\delta(t, m, \sigma) = \text{network}(m)$. If m is totally ordered, we may write $\delta(t, m, \sigma) = \text{subtasks}(m)$.

Note

For planning, we will interested in finding method instances that are both applicable in the current state and relevant for some task we are trying to accomplish.

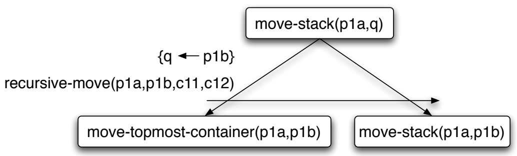
Applicale and Relevant Method Example

Example (Applicable and Revelant Method)

Let t be the nonprimitive task move-stack(p1a,q), s the state of the world show in previous slide, and m be the method instance recursive-move(p1a,p1b,c11,c12). m is applicable to s, revelant for t under substitution $\sigma = \{\text{q} \leftarrow \text{p1b}\}$, and decomposes t into:

$$\delta(t, m, \sigma) = \langle\text{move-topmost-container(p1a,p1b),move-stack(p1a,p1b)}\rangle$$

- Graphical representation of the method decomposition:



STN Planning Domain Definition

Definition (STN Planning Domain)

An STN planning domain is a pair

$$\mathcal{D} = (O, M)$$

where

- O is a set of operators
- M is a set of methods.

$\mathcal{D}$ is a total-order planning domain if every $m \in M$ is totally ordered.

STN Planning Problem Definition

Definition (STN Planning Problem)

An STN planning problem is a 4-tuple

$$\mathcal{P} = (s_0, w, O, M)$$

where

- s_0 is the initial state
- w is a task network called the initial task network
- $\mathcal{D} = (O, M)$ is a STN planning domain

$\mathcal{P}$ is a total-order planning problem if w and D are totally ordered.

Solution Plan

Definition (Solution Plan)

Let $\mathcal{P} = (s_0, w, O, M)$ be a planning problem. Here are the cases in which a plan $\pi = \langle a_1, \ldots, a_n \rangle$ is solution for $\mathcal{P}$:

- **Case 1:** w is empty. Then π is a solution for $\mathcal{P}$ is π is empty, i.e., $\pi = \langle\rangle$.
- **Case 2:** There is a primitive task node $u \in w$ that has no predessors in w. Then π is a solution for $\mathcal{P}$ is a_1 is applicable to t_u in s_0 and the plan $\pi = \langle a_2, \ldots, a_n \rangle$ is a solution of the planning problem:
$$\mathcal{P}' = (\gamma(s_0, a_1), w - \{u\}, O, M)$$
- **Case 3:** There is a nonprimitive task node $u \in w$ that has no predessor in w. Suppose there is an instance m of some method in M such that m is revelant for t_u and applicable in s_0. Then π is a solution for $\mathcal{P}$ is there is a task network $w' \in \delta(w, u, m, \sigma)$ such that π is a solution for (s_0, w', O, M).

Solution Plan Example (1/2)

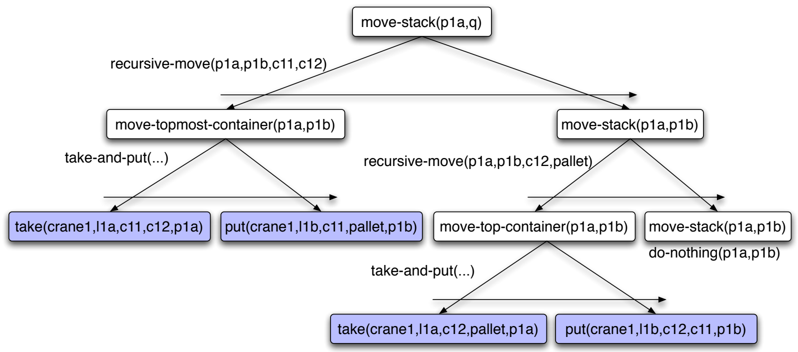
Example (DWR Solution Plan)

Let $\mathcal{P} = (s_0, w, O, M)$, where s_0 is the state initial state of the DWR problem, $w = \langle$move-stack(p1a,p1b)$\rangle$, O is the usual set of operators, and M is the set of methods. Then there is only one solution for $\mathcal{P}$:

$$\begin{aligned}\pi \quad = \quad & \langle \text{take(crane1,l1a,c11,c12,p1a)}, \\ & \text{put(crane1,l1b,c11,pallet,p1b)}, \\ & \text{take(crane1,l1a,c12,pallet,p11)}, \\ & \text{put(crane1,l1b,c12,c11,p1b)} \rangle\end{aligned}$$

Solution Plan Example (2/2)

- Example of tree decomposition for the solution plan π:



Total-Order STN Planning

Total-order Forward Decomposition

Algorithm (TFD($s, \langle t_1, \ldots, t_k \rangle, O, M$))

if $k = 0$ **then return** *an empty plan* $\pi = \langle\rangle$
else 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 revelant 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 $\pi =$ Failure **then return** Failure
  else return $a \cdot \pi$
else if t_1 *is nonprimitive* **then**
  active $\leftarrow \{(m, \sigma) \mid$ *m is a ground instance of a method in* M, σ *is a substitution such that m is revelant 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)*$\cdot\sigma(\langle t_2, \ldots, t_k \rangle)$
  return TFD(s, w, O, M)

TFD Comparaison

1. Like Forward-search, TFD considers only actions whose preconditions are satisfied in the current state. Moreover, like Backward-search, it considers only operators that revelant for the task to achieve
   $\Rightarrow$ greatly increase the efficiency of the search
2. Like Forward-search, TFD generates actions in the same order in which they will be executed
   $\Rightarrow$ it knows the current state of the world

Partial-Order STN Planning

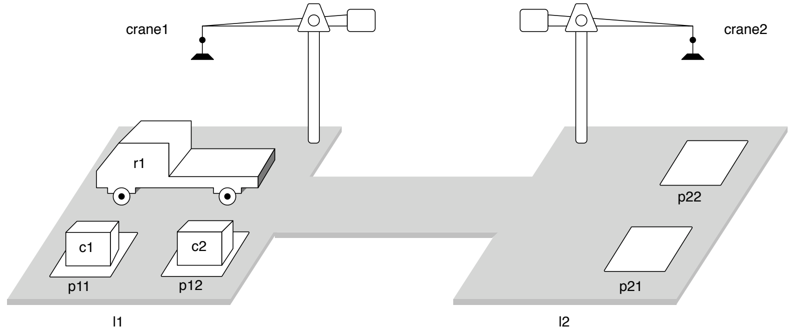
Partial-Order STN Planning

Why partial-order planning is interested to be considered ?

⇒ because not all planning domains can be rewritten into total-order planning

Partial-Order STN Planning: Example (1/5)

- Consider the following initial state for the DWR domain:



Partial-Order STN Planning: Example (2/5)

Example (DWR methods to move two containers at once)

transfer2(*c1*,*c2*,*l1*,*l2*,*r*) *;; method to transfert c1 and c2*

  task: transfer-two-containers(*c1*,*c2*,*l1*,*l2*,*r*)

  precond: *;; no preconditions*

  subtasks: ⟨transfer-one-container(*c1*,*l1*,*l2*,*r*), transfer-one-container(*c2*,*l1*,*l2*,*r*)⟩

transfer1(*c*,*l1*,*l2*,*r*) *;; method to transfert c*

  task: transfer-one-container(*c*,*l1*,*l2*,*r*)

  precond: *;; no preconditions*

  network: u_1 = setup(*c*,*r*), u_2 = move-robot(*l1*,*l2*), u_3 = finish(*c*,*r*), $\{(u_1, u_2), (u_2, u_3)\}$

move1(*r*,*l1*,*l2*) *;; method to move r if r is not at l2*

  task: move-robot(*l1*,*l2*)

  precond: at(*r*,*l1*)

  subtasks: ⟨move(*r*,*l1*,*l2*)⟩

Partial-Order STN Planning: Example (3/5)

Example (DWR methods to move two containers at once)

move0(*r*,*l1*,*l2*) *;; method to move r if r is already at l2*

  task: move-robot(*l1*,*l2*)

  precond: at(*r*,*l2*)

  subtasks: ⟨⟩ *;; no subtasks*

do-setup(*c*,*d*,*k*,*l*,*p*,*r*) *method to prepare for moving a container*

  task: setup(*c*,*r*)

  precond: on(*c*,*d*), in(*c*,*p*), belong(*k*,*l*), attached(*p*,*l*), at(*r*,*l*)

  network: u_1 = take(*k*,*l*,*c*,*r*), u_2 = put(*k*,*l*,*c*,*d*,*p*), $\{(u_1, u_2)\}$

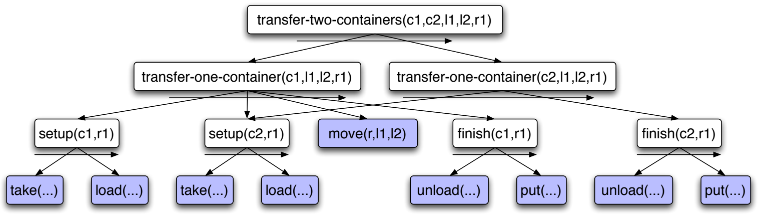
unload-robot(*c*,*d*,*k*,*l*,*p*,*r*) *;; method to finish after moving a container*

  task: finish(*c*,*r*)

  precond: attached(*p*,*l*), loaded(*r*,*c*), top(*d*,*p*), belong(*k*,*l*), at(*r*,*l*)

  network: u_1 = unload(*k*,*l*,*c*,*r*), u_2 = put(*k*,*l*,*c*,*d*,*p*), $\{(u_1, u_2)\}$

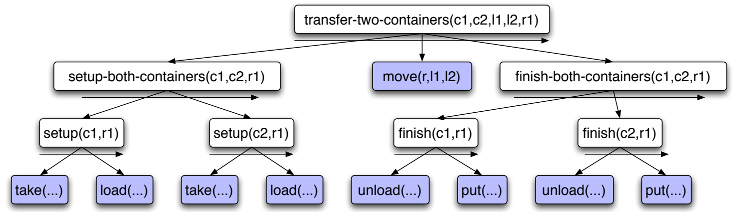
Partial-Order STN Planning: Example (4/5)



Interleaved Decomposition Tree

- The subtasks of the root are unordered, and their subtasks are interleaved
- Decomposition tree like this cannot occur in total-order STN planning domain

Partial-Order STN Planning: Example (5/5)



Noninterleaved Decomposition Tree

- To obtain a totally ordered tree, the best is to write method that generate a noninterleaved decomposition tree

Partial-order Forward Decomposition

Algorithm (PFD(s, w, O, M))

if $w = \emptyset$ **then return** *an empty plan* $\pi = \langle\rangle$
nondeterministically choose any $u \in w$ *that as no predessors in* w
else if t_1 *is primitive task* **then**
  active $\leftarrow \{(a, \sigma) \mid$ *a is a ground instance of an operator in O, σ is a substitution such that a is revelant for* $\sigma(t_1)$, *and a is applicable to s* $\}$
  if active $= \emptyset$ **then return** Failure
  nondeterministically choose any $(a, \sigma) \in$ active
  $\pi \leftarrow \mathrm{PFD}(\gamma(s, a), \sigma(w - \{u\}), O, M)$
  if $\pi =$ Failure **then return** Failure
  else return $a \cdot \pi$
else if t_1 *is nonprimitive* **then**
  active $\leftarrow \{(m, \sigma) \mid$ *m is a ground instance of a method in M, σ is a substitution such that m is revelant for* $\sigma(t_1)$, *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 $\mathrm{PFD}(s, w', O, M)$

HTN STN Planning

HTN Planning

- In STN planning, two kinds of constraints are associated with a method:
  1. preconditions
  2. ordering constraints
- Ordering constraints are explicitely represented in the task network but not preconditions
- HTN planning is a generalization of SNT planning that give the planning procedure more freedom about how to construct the task network

Task Network Definition

Definition

A task network is a pair

$$w = (U, C)$$

where

- U is a set of task nodes and
- C is a set of constraints.

Task Network Constraints

- HTN Task Network can handle the following kinds of constraints:
  1. A precedence constraint is an expression of the form $u \prec v$, where u and v are task node. Its meaning is identical to the edge (u, v) in STN planning.
  2. A before-constraint is a generalization of the notion of a precondition in STN planning. It is a constraint of the form before(U', l), where $U' \subseteq U$ is a set of task nodes and l is a literal.

Example

For instance, consider the task u is a task node for which t_u = move(r2,l2,l3). Then the constraints before($\{u\}$, at(r2,l2)) says that r2 must be at l2 just before we move it from l2 to l3.

  3. An after-constraint has the form after(U', l). It is like a before-constraint except that it says that l must be true in the state that occurs just after last (U', π)
  4. A between-constraint has the form between(U', U'', l). It says that literal l must be true in the state just after last (U', π), the state just before first (U'', π) and all of the states in between

HTN Methods: Definition

Definition (HTN Method)

An HTN method is a 4-tuple

$$m = (name(m), task(m), subtasks(m), constr(m))$$

in which the elements are described as follows:

- $name(m)$, the name of the method, i.e., a expression if the form $m(x_1, \dots, x_2)$ where n is an unique method symbol and $x_1, \dots, x_2$ are all of the variables symbols that occurs anywhere in m
- $task(m)$ is a non primitive task
- $(subtasks(m), constr(m))$ is a task network

Dynamic of HTN Methods

Suppose that $w = (U, C)$ is a task network, $u \in U$ is a task node, t_u is it task, m is an instance of a method in M, and task$(m) = t_u$. Then m decomposes u into subtasks(m'), producing the task network:

$$\delta(w, u, m) = ((U - \{u\}) \cup subtasks(m'), C' \cup constr(m'))$$

where C' is the following modified version of C:

- For every precedence constraint that constains u, replace it with precedence constraints containing the node of $subtasks(m')$

Example

If $subtasks(m') = \{u_1, u_2\}$, then we would replace $u \prec v$ with $u_1 \prec v$ and $u_2 \prec v$

- For every before, after, between constraints in which there is a set of task nodes U' that contains u, replace U' with $(U' - \{u\}) \cup subtasks(m')$

Example

If $subtasks(m') = \{u_1, u_2\}$, then we would replace before$(\{u, v\}, l)$ with before$(\{u_1, u_2, v\}, l)$

HTN Methods: Example (1/2)

Example (DWR HTN Methods of example slide 321)

transfer2(*c1*,*c2*,*l1*,*l2*,*r*) *;; method to move c1 and c2 from pile p1 to pile p2*

task: transfer-two-containers(*c1*,*c2*,*l1*,*l2*,*r*)
subtasks: u_1 = transfer-one-container(*c1*,*l1*,*l2*,*r*), u_2 = transfer-one-container(*c2*,*l1*,*l2*,*r*)
constr: $u_1 \prec u_2$

transfer1(*c*,*l1*,*l2*,*r*) *;; method to transfert c*

task: transfer-one-container(*c*,*l1*,*l2*,*r*)
subtasks: u_1 = setup(*c*,*r*), u_2 = move-robot(*l1*,*l2*), u_3 = finish(*c*,*r*)
constr: $u_1 \prec u_2$ $u_2 \prec u_3$

move1(*r*,*l1*,*l2*) *;; method to move r if r is not at l2*

task: move-robot(*l1*,*l2*)
subtasks: move(*r*,*l1*,*l2*)
constr: before($\{u_1\}$, at(*r*,*l1*))

HTN Methods: Example (2/2)

Example (DWR HTN Methods of example slide 321)

move0(*r*,*l1*,*l2*) *;; method to move r if r is already at l2*

task: move-robot(*l1*,*l2*)
subtasks: *;; no subtasks*
constr: before($\{u_0\}$, at(*r*,*l2*))

do-setup(*c*,*d*,*k*,*l*,*p*,*r*) *method to prepare for moving a container*

task: setup(*c*,*r*)
subtasks: u_1 = take(*k*,*l*,*c*,*r*), u_2 = put(*k*,*l*,*c*,*d*,*p*)
network: $u_1 \prec u_2$, before($\{u_1\}$, on(*c*,*d*)), before($\{u_1\}$, attached(*p*,*l*)), before($\{u_1\}$, in(*c*,*p*)), before($\{u_1\}$, belong(*k*,*l*)), before($\{u_1\}$, at(*r*,*l*))

unload-robot(*c*,*d*,*k*,*l*,*p*,*r*) *;; method to finish after moving a container*

task: finish(*c*,*r*)
subtasks: u_1 = unload(*k*,*l*,*c*,*r*), u_2 = put(*k*,*l*,*c*,*d*,*p*)
network: $u_1 \prec u_2$, before($\{u_1\}$, attached(*p*,*l*)), before($\{u_1\}$, loaded(*r*,*c*)), before($\{u_1\}$, top(*d*,*p*)), before($\{u_1\}$, belong(*k*,*l*)), before($\{u_1\}$, at(*r*,*l*))

HTN Planning Domain and Problem Definition

Definition (HTN Planning Domain)

An HTN planning domain is a pair $\mathcal{D} = (O, M)$ where

- O is a set of operators
- M is a set of methods.

Definition (HTN Planning Problem)

An HTN planning problem is a 4-tuple $\mathcal{P} = (s_0, w, O, M)$ where

- s_0 is the initial state
- w is a task network called the initial task network
- $\mathcal{D} = (O, M)$ is a STN planning domain

HTN Solution Plan Definition (1/2)

Definition (HTN Solution Plan)

- **Case 1:** If $w = (U, C)$ is primitive, then a plan $\pi = \langle a_1, \ldots, a_k \rangle$ is a solution for $\mathcal{P}$ if there is a ground instance (U', C') of (U, C) and a total ordering $\langle u_1, \ldots, u_k \rangle$ of the node U' such that all the following condition hold:
  1. The action in π are the ones named by the node $u_1, \ldots, u_k$, i.e., $\text{name}(a_i) = t_{u_i}$ for $i = 1, \ldots k$
  2. The plan π is executable from s_0
  3. The total ordering $\langle u_1, \ldots, u_k \rangle$ satisfies the precedence constraints in C', i.e., C' contains no constraint $u_i \prec u_j$ such that $j \leq i$
  4. For every constraints before(U', l) in C', l holds in the state s_{i-1} that immediately precedes action a_i, where a_i is the action named by the first node of U'.
  5. For every constraints after(U', l) in C', l holds in the state s_j produced by the action a_j, where a_j is the action named by the last node of U'.
  6. For every constraints between(U', U'', l) in C', l holds in every state that comes between a_i and a_j, where a_i is the action named by the last node of U' and a_j the action named by the first node of U''.

HTN Solution Plan Definition (2/2)

Definition (HTN Solution Plan)

- **Case 2:** If $w = (U, C)$ is nonprimitive, (i.e., al least one task in U is nonprimitive), then a plan π is a solution for $\mathcal{P}$ if there is a sequence of task decompositions that can be applied to w to produce primitive task network w' such taht π is a solution for w'. In this case, the decomposition tree for π is the tree structure corresponding to these task decompositions.

HTN Planning Procedure

Algorithm (Abstract-HTN(s, U, C, O, M))

if (U, C) *can be shown to have no solution* **then return** Failure
else if U *is primitive* **then**
  if (U, C) *has no solution* **then return** Failure
  else **return** *nondeterministically a plan* π *from any such solution*
else
  choose a nonprimitive task node $u \in U$
  active $\leftarrow \{m \in M \mid task(m) \text{ is unifiable with } t_u\}$
  if active $\neq \emptyset$ **then** *nondeterministically choose any* $m \in$ active
  $\sigma \leftarrow$ *an mgu for* m *and* t_u *that renames all variables of* m
  $(U', C') \leftarrow \delta(\sigma(U, C), \sigma(u), \sigma(m))$
  return `Abstract-HTN`(s, U', C', O, M)
end

Comparaison and extensions of HTN Planning

HTN versus Classical Planning

- STN planning and thus HTN planning can be used to encode undecidable problem, but not classical planning
- However STN and HTN language can produce undesirable effects

Example (Recursive method calls)

method1()

task: task1()
precond: *;; no preconditions*
subtasks: op1(),
task1(), op2()

op1()

precond: *;; no preconditions*
effects: *;; no effects*

method2()

task: task1()
precond: *;; no preconditions*
subtasks: *;; no subtasks*

op2()

precond: *;; no preconditions*
effects: *;; no effects*

The solutions to this problem are as follows:

$$\pi_0 = \langle\rangle, \pi_1 = \langle \text{op1}(), \text{op2}() \rangle, \pi_2 = \langle \text{op1}(), \text{op1}(), \text{op2}(), \text{op2}() \rangle$$

Complexity of plan existance for HTN planning

Restrictions on nonprimitive tasks	Must the HTNs be totally ordered ?	Are variables allowed?	
		No	Yes
None	No Yes	Undecidable[a] In exptime pspace-hard	Undecidable[a,b] in dexptime[d] expspace-hard
"Regularity" (≤ 1 nonprimitive task, which must follow all primitive tasks)	Does not matter	pspace-complete	expspace-complete[c]
No nonprimitive tasks	No Yes	NP-complete Polynomial time	NP-complete NP-complete

[a] Decidable if we impose acyclic restrictions
[b] Undecidable even when the planning domain is fixed in advance
[c] In pspace when the planning domain is fixed in advance, and pspace-complete for some fixed planning domains
[d] dexptime means double-exponential time

HTN Planning Extensions

- The main extensions of HTN planning are:
  1. Function Symbols. If we allow the planning language to contain function symbols, then aguments of an atom, or task are no longer restricted to being constant symbol of variable symbols.
  2. Axioms. To incorporate axiomatic inference, we will need to used theorem prover as a subroutine of the planning procedure.
  3. Attached Procedures. We can modify the precondition evaluation algorithm to recognize that certain terms or predicate symbols are to be evaluated by using attached procedure rather that by using the normal theorem prover.
  4. Time. It is possible to generalize PFD and Abstract-HTN to certain kinds of temporal planning, e.g., to deal with action that have time durations and may overlap with each other.

To go further

Exercices

Exercice 1

Write totally ordered methods to generate the noninterleaved decomposition tree similar to the one shown slide 344.

Exercice 2

Suppose we write a deterministic implementation of TFD that does a depth-first search of its decomposition tree. Is this implementation complete ? Why or why not ?

Exercice 3

In example slide 329, suppose we allow the initial state to contain an atom need-to-move(p,q) for each stack of the containers that needs to be moved from som pile p to some other q. Rewrite the methods and operators so that instead of being restricted to work on three stacks of containers, they will work correctly for an arbitrary number of stacks and containers.

Further readings

- P. Bercher, R. Alford, D. Höller:
  A Survey on Hierarchical Planning - One Abstract Idea, Many Concrete Realizations.
  IJCAI 2019: 6267-6275
- D. Holler, G. Behnke, P. Bercher, S. Biundo, H. Fiorino, D. Pellier, R. Alford:
  HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems.
  AAAI 2020: 9883-9891